

Algebra delle Stringhe e Linguaggi Formali

La teoria dei linguaggi formali rappresenta uno dei pilastri fondamentali dell'informatica teorica, fornendo gli strumenti matematici necessari per comprendere la struttura, le proprietà e le limitazioni dei sistemi di computazione. L'approccio algebrico a questa disciplina permette di trascendere la mera descrizione operativa dei linguaggi e degli automi, rivelando le profonde connessioni tra le strutture linguistiche e le strutture algebriche classiche. Questo trattato si propone di esplorare in modo esaustivo e rigoroso le fondamenta algebriche dei linguaggi formali, dalle proprietà elementari dei monoidi liberi alle sofisticate tecniche della teoria dei semigrupp sintattici, passando attraverso la caratterizzazione algebrica delle espressioni regolari e degli automi a stati finiti.

L'intuizione fondamentale che guida l'approccio algebrico è che le stringhe, intese come sequenze finite di simboli tratti da un alfabeto, formano naturalmente una struttura algebrica quando munite dell'operazione di concatenazione. Questa operazione, sebbene semplice nella sua definizione, dà origine a un ricco universo di proprietà e teoremi che illuminano la natura intrinseca dei linguaggi formali. La concatenazione non è commutativa, fatto che distingue i monoidi di stringhe dalle strutture algebriche più familiari come i gruppi abeliani, ma è associativa e possiede un elemento neutro, la stringa vuota, che gioca un ruolo analogo allo zero nell'addizione o all'uno nella moltiplicazione.

Monoidi Liberi e la Struttura Algebrica delle Stringhe

Alfabeti, Stringhe e Concatenazione

Iniziamo con le definizioni fondamentali che costituiscono il linguaggio della teoria. Un alfabeto Σ è un insieme finito non vuoto di simboli, detti anche caratteri o lettere. Non vi è alcuna restrizione sulla natura degli elementi di Σ : possono essere lettere dell'alfabeto latino, cifre numeriche, simboli speciali, o qualsiasi altro oggetto distinguibile. L'unico requisito è che l'alfabeto sia finito, condizione che riflette la natura discreta e finita dei sistemi di computazione reali.

Una stringa o parola su un alfabeto Σ è una sequenza finita di elementi di Σ . Formalmente, una stringa di lunghezza n è una funzione $w: \{1, 2, \dots, n\} \rightarrow \Sigma$, ma nella pratica si rappresenta semplicemente elencando i simboli in sequenza, come $w = a_1 a_2 \dots a_n$ dove ciascun a_i appartiene a Σ . La lunghezza di una stringa w , denotata con $|w|$, è il numero di simboli che la compongono, contati con la loro molteplicità. Esiste una stringa particolare di lunghezza zero, chiamata stringa vuota e denotata con ε , che non contiene alcun simbolo. L'esistenza della stringa vuota è cruciale per la struttura algebrica che stiamo per definire.

L'insieme di tutte le stringhe di lunghezza esattamente n su un alfabeto Σ si denota con Σ^n . Questo insieme ha cardinalità $|\Sigma|^n$, come si può facilmente verificare per induzione: ogni posizione nella stringa può essere riempita in $|\Sigma|$ modi diversi, e vi sono n posizioni

indipendenti. L'insieme di tutte le stringhe finite su Σ , di qualsiasi lunghezza inclusa zero, si denota con Σ^* e si chiama chiusura di Kleene di Σ . Formalmente, $\Sigma^* = \bigcup \{n \geq 0\} \Sigma^n$. *Talvolta si considera anche l'insieme $\Sigma^+ = \bigcup \{n \geq 1\} \Sigma^n$, che esclude la stringa vuota e si chiama chiusura positiva di Σ . È chiaro che $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.*

La concatenazione è l'operazione binaria fondamentale sulle stringhe. Date due stringhe $u = a_1 a_2 \dots a_n$ e $v = b_1 b_2 \dots b_m$, la loro concatenazione uv è la stringa ottenuta giustapponendo i simboli di v dopo quelli di u , ovvero $uv = a_1 a_2 \dots a_n b_1 b_2 \dots b_m$. La lunghezza della concatenazione è la somma delle lunghezze: $|uv| = |u| + |v|$. La stringa vuota ε si comporta come elemento neutro rispetto alla concatenazione: per ogni stringa w si ha $\varepsilon w = w\varepsilon = w$. Questa proprietà è immediata dalla definizione, ma il suo significato algebrico è profondo.

Il Monoide Libero Σ^*

L'insieme Σ^* munito dell'operazione di concatenazione forma una struttura algebrica chiamata monode libero generato da Σ . Un monode è un insieme M dotato di un'operazione binaria associativa $\cdot : M \times M \rightarrow M$ e di un elemento neutro $e \in M$ tale che $e \cdot x = x \cdot e = x$ per ogni x in M . Nel nostro caso, $M = \Sigma^*$, l'operazione è la concatenazione, e l'elemento neutro è ε .

L'associatività della concatenazione è una proprietà fondamentale che merita una dimostrazione esplicita. Date tre stringhe u, v, w , vogliamo mostrare che $(uv)w = u(vw)$. Se $u = a_1 \dots a_n$, $v = b_1 \dots b_m$, e $w = c_1 \dots c_k$, allora $(uv)w = (a_1 \dots a_n b_1 \dots b_m) c_1 \dots c_k = a_1 \dots a_n b_1 \dots b_m c_1 \dots c_k$ per definizione di concatenazione. Analogamente, $u(vw) = a_1 \dots a_n (b_1 \dots b_m c_1 \dots c_k) = a_1 \dots a_n b_1 \dots b_m c_1 \dots c_k$. Le due espressioni coincidono, il che dimostra l'associatività. Questa proprietà permette di scrivere concatenazioni multiple senza ambiguità, omettendo le parentesi: $u_1 u_2 u_3 \dots u_n$ ha un significato ben definito indipendentemente dall'ordine in cui si effettuano le concatenazioni.

Un aspetto cruciale del monode Σ^* è la sua non commutatività. In generale, per stringhe u e v , non vale $uv = vu$. Per esempio, se $\Sigma = \{a, b\}$, allora $ab \neq ba$. Questa non commutatività riflette il fatto che l'ordine dei simboli in una stringa è significativo e non può essere permutato liberamente. Esistono tuttavia casi speciali in cui due stringhe commutano: se u e v sono potenze di una stessa stringa w , ovvero $u = w^m$ e $v = w^n$ per qualche w e interi m, n , allora $uv = w^{m+n} = vu$. Questo fatto è alla base di importanti risultati nella teoria combinatoria delle stringhe.

Il termine "libero" nella denominazione "monode libero" ha un significato tecnico preciso nella teoria delle categorie e dell'algebra universale. Intuitivamente, significa che Σ^* è il monode "più generale" che si può costruire a partire dall'insieme Σ , senza imporre alcuna relazione tra gli elementi oltre a quelle imposte dagli assiomi dei monoidi. Più formalmente, il monode libero Σ^* soddisfa la seguente proprietà universale: data qualsiasi funzione $f: \Sigma \rightarrow M$ dove M è un monode, esiste un unico omomorfismo di monoidi $f^*: \Sigma^* \rightarrow M$ che estende f , definito da $f^*(a_1 a_2 \dots a_n) = f(a_1) \cdot f(a_2) \cdot \dots \cdot f(a_n)$ in M . Questa proprietà universale caratterizza Σ^* in modo completo, a meno di isomorfismo.

Sottomonoidi e Linguaggi

Un linguaggio su un alfabeto Σ è semplicemente un sottoinsieme $L \subseteq \Sigma^*$. Non ogni linguaggio forma un sottomonoido di Σ^* , ma quelli che lo fanno hanno proprietà particolari. Un sottoinsieme L di Σ^* è un sottomonoido se soddisfa due condizioni: primo, contiene l'elemento neutro ε ; secondo, è chiuso rispetto alla concatenazione, ovvero se $u, v \in L$ allora $uv \in L$.

Un esempio fondamentale di sottomonoido è il linguaggio generato da un sottoinsieme X di Σ^* . Questo è il più piccolo sottomonoido di Σ^* che contiene X , e si denota con X^* . Concretamente, X^* consiste di tutte le stringhe che possono essere ottenute concatenando un numero finito (possibilmente zero) di elementi di X . Formalmente, $X^* = \{x_1x_2\dots x_n : n \geq 0, x_i \in X \text{ per ogni } i\}$. Per convenzione, la concatenazione di zero elementi dà la stringa vuota ε . Se $X = \{a, b\}$, allora $X^* = \{a, b\}^*$, che coincide con Σ^* se $\Sigma = \{a, b\}$. Se $X = \{ab, ba\}$, allora X^* consiste di tutte le stringhe che si possono ottenere alternando e concatenando ab e ba in qualsiasi ordine e numero.

L'operazione $*$ applicata a un linguaggio si chiama stella di Kleene o chiusura di Kleene. È facile verificare che L^* è sempre un sottomonoido di Σ^* , qualunque sia L . Inoltre, L^* è il più piccolo sottomonoido contenente L , nel senso che se M è un sottomonoido e $L \subseteq M$, allora $L^* \subseteq M$. Questa è una conseguenza della definizione: ogni elemento di L^* è una concatenazione di elementi di L , quindi per la chiusura di M rispetto alla concatenazione, appartiene a M .

Un altro concetto importante è quello di linguaggio chiuso positivamente, denotato con L^+ . Questo consiste di tutte le concatenazioni di uno o più elementi di L , escludendo quindi ε a meno che ε non appartenga già a L . Formalmente, $L^+ = \{x_1x_2\dots x_n : n \geq 1, x_i \in L \text{ per ogni } i\}$. La relazione tra L^* , L^+ e L è data da $L^* = L^+ \cup \{\varepsilon\}$ e $L^+ = LL^*$.

Morfismi di Monoidi e Omofonia

Un omomorfismo o morfismo tra due monoidi $(M_1, \cdot_1, e_1)$ e $(M_2, \cdot_2, e_2)$ è una funzione $h: M_1 \rightarrow M_2$ che preserva la struttura del monoido, ovvero soddisfa $h(e_1) = e_2$ e $h(x \cdot_1 y) = h(x) \cdot_2 h(y)$ per ogni x, y in M_1 . Nel contesto dei monoidi di stringhe, un morfismo $h: \Sigma^* \rightarrow \Gamma^*$ è completamente determinato dai suoi valori sugli elementi di Σ , grazie alla proprietà di libertà discussa in precedenza. Se conosciamo $h(a)$ per ogni $a \in \Sigma$, possiamo estendere h a tutto Σ^* mediante $h(a_1a_2\dots a_n) = h(a_1)h(a_2)\dots h(a_n)$.

I morfismi di monoidi sono strumenti fondamentali nella teoria dei linguaggi formali. Un morfismo $h: \Sigma^* \rightarrow \Gamma^*$ induce naturalmente operazioni sui linguaggi. Se $L \subseteq \Sigma^*$ è un linguaggio, l'immagine diretta di L tramite h è $h(L) = \{h(w) : w \in L\} \subseteq \Gamma^*$. Se $K \subseteq \Gamma^*$ è un linguaggio, l'immagine inversa di K tramite h è $h^{-1}(K) = \{w \in \Sigma^* : h(w) \in K\} \subseteq \Sigma^*$. È importante notare che $h^{-1}(K)$ è ben definito per ogni K , anche se h non è iniettivo, perché stiamo considerando l'immagine inversa insiemistica, non l'inversa funzionale.

Un morfismo particolare di grande importanza è il morfismo di cancellazione o proiezione. Se $\Sigma = \Sigma_1 \cup \Sigma_2$ con $\Sigma_1 \cap \Sigma_2 = \emptyset$, possiamo definire un morfismo $\pi_1: \Sigma^* \rightarrow \Sigma_1^*$ che "cancella" tutti i simboli di Σ_2 . Formalmente, $\pi_1(a) = a$ se $a \in \Sigma_1$ e $\pi_1(a) = \varepsilon$ se $a \in \Sigma_2$. Questo morfismo estende quindi a $\pi_1(w) =$ la stringa ottenuta da w rimuovendo tutti i simboli in Σ_2 .

Un morfismo si dice iniettivo se $h(u) = h(v)$ implica $u = v$, ovvero stringhe diverse hanno immagini diverse. Si dice suriettivo se per ogni $w \in \Gamma^*$ esiste almeno un $u \in \Sigma^*$ tale che $h(u) = w$. Un morfismo che è sia iniettivo che suriettivo è un isomorfismo, e in questo caso i due monoidi sono essenzialmente identici dal punto di vista algebrico. Un morfismo si dice non cancellante se $h(a) \neq \varepsilon$ per ogni $a \in \Sigma$, ovvero nessun simbolo viene mappato sulla stringa vuota. I morfismi non cancellanti preservano la lunghezza delle stringhe nel senso che $|h(w)|$ è sempre almeno $|w|$ volte la lunghezza minima delle immagini dei simboli.

Un caso particolare interessante è quello dei morfismi alfabetici o sostituzioni uniformi, dove $h(a)$ ha la stessa lunghezza k per ogni $a \in \Sigma$. In questo caso, $|h(w)| = k|w|$ per ogni w , e il morfismo è iniettivo se e solo se è iniettivo sui singoli simboli. Questi morfismi sono importanti nello studio dei codici e della compressione.

Relazioni di Equivalenza e Congruenze

Nel monoide Σ^* , possiamo definire varie relazioni di equivalenza che catturano differenti nozioni di "somiglianza" tra stringhe. Una relazione di equivalenza su Σ^* è una relazione binaria $\sim$ che è riflessiva ($w \sim w$ per ogni w), simmetrica (se $u \sim v$ allora $v \sim u$), e transitiva (se $u \sim v$ e $v \sim z$ allora $u \sim z$). Una relazione di equivalenza partiziona Σ^* in classi di equivalenza, dove due stringhe appartengono alla stessa classe se e solo se sono equivalenti.

Una congruenza su un monoide M è una relazione di equivalenza $\sim$ che è compatibile con l'operazione del monoide, nel senso che se $u \sim v$ e $x \sim y$, allora $ux \sim vy$. Nel caso di Σ^* , questo significa che la concatenazione di stringhe equivalenti produce risultati equivalenti. Le congruenze sono importanti perché permettono di definire quozienti del monoide: dato un monoide M e una congruenza $\sim$, l'insieme delle classi di equivalenza $M/\sim$ eredita naturalmente una struttura di monoide, con l'operazione $[u] \cdot [v] = [uv]$, dove $[w]$ denota la classe di equivalenza di w .

Un esempio fondamentale di congruenza è la congruenza sintattica di un linguaggio L , che verrà discussa in dettaglio più avanti. Per ora, introduciamo alcune congruenze più semplici. La congruenza di lunghezza è definita da $u \sim v$ se e solo se $|u| = |v|$. Questa è chiaramente una congruenza perché $|ux| = |u| + |x|$ e $|vy| = |v| + |y|$, quindi se $|u| = |v|$ e $|x| = |y|$ segue $|ux| = |vy|$. Il monoide quoziente $\Sigma^*/\sim$ è isomorfo al monoide additivo $(\mathbb{N}, +, 0)$ degli interi non negativi, dove la classe di equivalenza di una stringa di lunghezza n corrisponde all'intero n .

Un'altra congruenza importante è quella indotta da un morfismo. Se $h: \Sigma^* \rightarrow \Gamma^*$ è un morfismo, possiamo definire $u \sim_\square v$ se e solo se $h(u) = h(v)$. Questa è una congruenza perché h preserva la concatenazione: se $h(u) = h(v)$ e $h(x) = h(y)$, allora $h(ux) = h(u)h(x) = h(v)h(y) = h(vy)$. Il monoide quoziente $\Sigma^*/\sim_\square$ è isomorfo all'immagine di h , ovvero al sottomonoido $h(\Sigma^*)$ di Γ^* .

Sottoparole, Prefissi, Suffissi e Fattori

La struttura sequenziale delle stringhe permette di definire importanti relazioni d'ordine parziale. Una stringa u è un prefisso di una stringa w , scritto $u \leq_\square w$, se esiste una stringa v tale che $w = uv$. Dualmente, u è un suffisso di w , scritto $u \leq_\square w$, se esiste una stringa v tale

che $w = vu$. Più generalmente, u è un fattore o sottoparola di w se esistono stringhe v, z tali che $w = vuz$. Se u è un fattore di w e $u \neq w$, diciamo che u è un fattore proprio di w . Analogamente definiamo prefissi propri e suffissi propri.

Le relazioni di prefisso e suffisso sono ordini parziali sull'insieme Σ^* . La riflessività è ovvia ($w = \varepsilon w = w\varepsilon$). L'antisimmetria segue dal fatto che se $u \leq w$ e $w \leq u$, allora esistono v, z tali che $w = uv$ e $u = wz$, da cui $u = uvz$, che implica $vz = \varepsilon$ (perché la concatenazione è cancellabile a sinistra), quindi $v = z = \varepsilon$ e $u = w$. La transitività è immediata: se $u \leq v$ e $v \leq w$, allora $v = ux$ e $w = vy$ per qualche x, y , da cui $w = uxy$ e $u \leq w$.

L'insieme dei prefissi di una stringa w forma una catena totalmente ordinata rispetto a $\leq$. Se u e v sono entrambi prefissi di w , allora uno è necessariamente prefisso dell'altro. Infatti, se $w = ux = vy$, possiamo assumere senza perdita di generalità che $|u| \leq |v|$. Allora u è un prefisso di v , perché v deve iniziare con gli stessi simboli di u (essendo entrambi prefissi di w), e poiché $|u| \leq |v|$, abbiamo che i primi $|u|$ simboli di v formano esattamente u .

Il numero di prefissi di una stringa di lunghezza n è esattamente $n + 1$, includendo la stringa vuota e la stringa stessa. Analogamente per i suffissi. Il numero di fattori è invece al più $(n + 1)(n + 2) / 2$, che si ottiene sommando i numeri di fattori di ogni lunghezza possibile: 1 fattore di lunghezza 0 (ε), n fattori di lunghezza 1, al più $n - 1$ fattori di lunghezza 2, eccetera, fino a 1 fattore di lunghezza n . La somma è $1 + 2 + \dots + (n + 1) = (n + 1)(n + 2) / 2$. Questo limite superiore è stretto per stringhe senza ripetizioni di fattori.

Potenze di Stringhe e Periodicità

Data una stringa w e un intero non negativo n , definiamo la potenza n -esima di w , denotata w^n , come la concatenazione di n copie di w . Formalmente, $w^0 = \varepsilon$, e $w^{n+1} = w^n w$. Le potenze di stringhe soddisfano le familiari leggi degli esponenti: $w^m w^n = w^{m+n}$ e $(w^m)^n = w^{mn}$. Queste proprietà seguono direttamente dalla definizione e dall'associatività della concatenazione.

Una stringa u si dice primitiva se non può essere scritta come potenza di un'altra stringa, ovvero se $u = v^n$ con $n \geq 1$ implica $n = 1$. Ogni stringa non vuota può essere scritta in modo unico come potenza di una stringa primitiva: se w è qualsiasi stringa, esiste un'unica stringa primitiva u e un unico intero $n \geq 1$ tale che $w = u^n$. Questa stringa u si chiama radice primitiva di w . Per esempio, la radice primitiva di $abababab$ è ab , poiché $abababab = (ab)^4$ e ab è primitiva.

Un teorema fondamentale nella teoria combinatoria delle stringhe è il lemma di Lyndon-Schützenberger, che caratterizza quando due potenze di stringhe si sovrappongono. Il lemma afferma che se $u^m = v^n$ con $m, n \geq 2$, allora u e v sono potenze di una stessa stringa, ovvero esiste w tale che $u = w^p$ e $v = w^q$ per qualche p, q . La dimostrazione di questo risultato si basa sull'analisi dei periodi delle stringhe.

Il periodo di una stringa w è un intero p tale che $w[i] = w[i + p]$ per ogni i dove entrambi i lati sono definiti (cioè $1 \leq i \leq |w| - p$). Il più piccolo periodo di w si chiama periodo minimo. Per esempio, la stringa $abababab$ ha periodo minimo 2. Il lemma della periodicità debole afferma che se una stringa w ha periodi p e q , e se $|w| \geq p + q - \gcd(p, q)$, dove $\gcd$ denota il massimo comun divisore, allora w ha anche periodo $\gcd(p, q)$. Questo risultato è ottimale nel

senso che esistono stringhe di lunghezza $p + q - \gcd(p, q) - 1$ che hanno periodi p e q ma non hanno periodo $\gcd(p, q)$.

Il Problema della Corrispondenza di Post

Un'applicazione profonda della teoria dei monoidi liberi è la caratterizzazione dell'indcidibilità del problema della corrispondenza di Post (PCP). Data una sequenza finita di coppie di stringhe $(u_1, v_1), (u_2, v_2), \dots, (u_n, v_n)$ su un alfabeto Σ , il PCP chiede se esiste una sequenza di indici $i_1, i_2, \dots, i_n$ (con ripetizioni permesse) tale che $u_{i_1}u_{i_2}\dots u_{i_n} = v_{i_1}v_{i_2}\dots v_{i_n}$.

Dal punto di vista algebrico, il PCP può essere formulato nei termini di morfismi. Definiamo due morfismi $h, g: \{1, 2, \dots, k\}^* \rightarrow \Sigma^*$ ponendo $h(i) = u_i$ e $g(i) = v_i$. Il PCP chiede se esiste una stringa non vuota $w \in \{1, 2, \dots, k\}^*$ tale che $h(w) = g(w)$, ovvero se i due morfismi coincidono su qualche stringa non vuota. Emil Post dimostrò nel 1946 che questo problema è indecidibile in generale, ovvero non esiste alcun algoritmo che, data un'istanza arbitraria del PCP, possa determinare in tempo finito se esiste una soluzione.

L'indcidibilità del PCP ha profonde conseguenze nella teoria dei linguaggi formali. Implica, per esempio, che il problema della equivalenza di grammatiche context-free è indecidibile, e che molti problemi naturali sui linguaggi formali non ammettono algoritmi di decisione. Questo limita fondamentalmente ciò che possiamo sperare di computare automaticamente sui linguaggi.

Espressioni Regolari come Algebra Booleana Estesa

Linguaggi Regolari e Operazioni

Un linguaggio regolare è un linguaggio che può essere descritto mediante un'espressione regolare, o equivalentemente, riconosciuto da un automa a stati finiti. Prima di definire formalmente le espressioni regolari, introduciamo le operazioni fondamentali sui linguaggi che costituiscono l'algebra dei linguaggi regolari.

Date due linguaggi $L_1, L_2 \subseteq \Sigma^*$, definiamo le seguenti operazioni:

L'unione $L_1 \cup L_2 = \{w : w \in L_1 \text{ oppure } w \in L_2\}$ è l'insieme delle stringhe che appartengono ad almeno uno dei due linguaggi. L'unione è un'operazione commutativa e associativa, con elemento neutro l'insieme vuoto $\emptyset$.

L'intersezione $L_1 \cap L_2 = \{w : w \in L_1 \text{ e } w \in L_2\}$ è l'insieme delle stringhe che appartengono ad entrambi i linguaggi. Anche l'intersezione è commutativa e associativa, ma ha come elemento neutro Σ^* , l'insieme di tutte le stringhe.

La concatenazione $L_1 L_2 = \{uv : u \in L_1 \text{ e } v \in L_2\}$ è l'insieme delle stringhe che possono essere ottenute concatenando una stringa di L_1 con una stringa di L_2 . La concatenazione è associativa ma non commutativa. L'elemento neutro è il linguaggio $\{\epsilon\}$ contenente solo la stringa vuota.

Il complemento di un linguaggio L rispetto a Σ^* è $\bar{L} = \Sigma^* \setminus L = \{w \in \Sigma^* : w \notin L\}$, l'insieme delle stringhe che non appartengono a L . Il complemento soddisfa le leggi di De Morgan: $(L_1 \cup L_2)^c = \bar{L}_1 \cap \bar{L}_2$ e $(L_1 \cap L_2)^c = \bar{L}_1 \cup \bar{L}_2$.

La stella di Kleene $L^* = \{w_1 w_2 \dots w_n : n \geq 0, w_i \in L \text{ per ogni } i\}$ è l'insieme di tutte le concatenazioni finite di stringhe in L , inclusa la concatenazione vuota ε . Abbiamo già discusso questa operazione nel contesto dei sottomonoidi.

Queste operazioni permettono di costruire una ricca struttura algebrica sull'insieme dei linguaggi. L'insieme $\mathcal{P}(\Sigma^*)$ di tutti i linguaggi su Σ^* , munito delle operazioni di unione e intersezione, forma un'algebra booleana. Gli elementi dell'algebra sono i linguaggi, l'unione corrisponde alla disgiunzione logica, l'intersezione alla congiunzione, e il complemento alla negazione. Gli elementi speciali sono $\emptyset$ (falso) e Σ^* (vero).

Definizione Formale delle Espressioni Regolari

Un'espressione regolare su un alfabeto Σ è una formula che denota un linguaggio regolare. La definizione è induttiva e procede come segue:

Le espressioni regolari di base sono: $\emptyset$, che denota il linguaggio vuoto; ε , che denota il linguaggio $\{\varepsilon\}$ contenente solo la stringa vuota; a per ogni $a \in \Sigma$, che denota il linguaggio $\{a\}$ contenente solo il simbolo a .

Se r e s sono espressioni regolari che denotano i linguaggi $L(r)$ e $L(s)$, allora:

- $(r + s)$ è un'espressione regolare che denota $L(r) \cup L(s)$, l'unione dei due linguaggi;
- (rs) o semplicemente rs è un'espressione regolare che denota $L(r)L(s)$, la concatenazione dei due linguaggi;
- (r^*) o semplicemente r^* è un'espressione regolare che denota $L(r)^*$, la stella di Kleene di $L(r)$.

Nient'altro è un'espressione regolare. Le parentesi possono essere omesse quando non vi è ambiguità, adottando la convenzione che la stella ha precedenza sulla concatenazione, che a sua volta ha precedenza sull'unione. Per esempio, $ab^* + c$ denota $(a(b^*)) + c$, ovvero il linguaggio $\{a\} \cdot \{\varepsilon, b, bb, bbb, \dots\} \cup \{c\} = \{a, ab, abb, abbb, \dots\} \cup \{c\}$.

È importante notare che $\emptyset$ e ε sono espressioni regolari distinte che denotano linguaggi diversi: $\emptyset$ denota l'insieme vuoto, mentre ε denota l'insieme contenente solo la stringa vuota. Analogamente, bisogna distinguere tra il simbolo a dell'alfabeto e l'espressione regolare a . La notazione a volte causa confusione, ma il contesto rende chiaro se ci si riferisce al simbolo o all'espressione.

L'Algebra delle Espressioni Regolari

Le espressioni regolari formano un sistema algebrico con le proprie leggi e identità. Due espressioni regolari r e s si dicono equivalenti, scritto $r \equiv s$, se denotano lo stesso linguaggio, ovvero se $L(r) = L(s)$. L'equivalenza di espressioni regolari è una congruenza rispetto alle operazioni di unione, concatenazione e stella.

Le seguenti identità valgono per tutte le espressioni regolari:

Idempotenza dell'unione: $r + r \equiv r$. Questa legge riflette il fatto che l'unione insiemistica è idempotente: $L \cup L = L$.

Commutatività dell'unione: $r + s \equiv s + r$. L'unione di linguaggi è commutativa.

Associatività dell'unione: $(r + s) + t \equiv r + (s + t)$. L'unione è associativa, quindi possiamo scrivere $r + s + t$ senza ambiguità.

Elemento neutro dell'unione: $r + \emptyset \equiv r$. Il linguaggio vuoto è neutro per l'unione.

Associatività della concatenazione: $(rs)t \equiv r(st)$. La concatenazione di linguaggi è associativa.

Elemento neutro della concatenazione: $\varepsilon r \equiv r\varepsilon \equiv r$. Il linguaggio $\{\varepsilon\}$ è neutro per la concatenazione.

Elemento assorbente della concatenazione: $\emptyset r \equiv r\emptyset \equiv \emptyset$. Il linguaggio vuoto è assorbente per la concatenazione.

Distributività della concatenazione rispetto all'unione: $r(s + t) \equiv rs + rt$ e $(r + s)t \equiv rt + st$. La concatenazione distribuisce sull'unione da entrambi i lati.

Proprietà della stella: $(r^*)^* \equiv r^*$. Applicare la stella due volte equivale ad applicarla una volta sola.

Espansione della stella: $r^* \equiv \varepsilon + rr^*$. Questa è l'equazione fondamentale della stella di Kleene, che esprime che $L^* = \{\varepsilon\} \cup LL^*$.

Assorbimento con la stella: $\varepsilon + r^* \equiv r^*$ e $rr \equiv r^*$. Queste identità riflettono che ε appartiene sempre a L^* e che $LL = L^*$.

Queste identità formano un sistema equazionale che caratterizza parzialmente l'algebra delle espressioni regolari. Tuttavia, questo sistema non è completo nel senso che esistono coppie di espressioni regolari equivalenti che non possono essere dimostrate equivalenti usando solo queste identità. La ricerca di un sistema di assiomi completo per l'equivalenza di espressioni regolari è un problema classico che ha portato a importanti sviluppi nella teoria.

Un risultato fondamentale è il teorema di Kleene, che afferma che un linguaggio è regolare (ovvero può essere descritto da un'espressione regolare) se e solo se può essere riconosciuto da un automa a stati finiti. Questo teorema crea un ponte tra la descrizione dichiarativa dei linguaggi mediante espressioni regolari e la descrizione operativa mediante automi, rivelando che le due prospettive hanno esattamente la stessa potenza espressiva.

Equazioni di Arden e Risoluzione di Sistemi

Un metodo sistematico per costruire espressioni regolari a partire da automi si basa sulla risoluzione di sistemi di equazioni lineari nell'algebra delle espressioni regolari. Questo metodo utilizza il lemma di Arden, un risultato fondamentale che caratterizza le soluzioni di equazioni della forma $X = AX + B$ dove A, B sono espressioni regolari e X è un'incognita.

Il lemma di Arden afferma che se $\varepsilon \notin L(A)$, allora l'equazione $X = AX + B$ ha come unica soluzione $X = AB$. *La dimostrazione si basa sull'osservazione che, formalmente, se X soddisfa $X = AX + B$, allora per sostituzione iterata otteniamo $X = AX + B = A(AX + B) + B = A^2X + AB + B = A^3X + A^2B + AB + B = \dots = A^nX + A^{n-1}B + \dots + AB + B$. Se $\varepsilon \notin L(A)$, ogni stringa in $L(A)$ ha lunghezza almeno uno, quindi le stringhe in $L(A^n)$ hanno lunghezza almeno n . Questo significa che per n sufficientemente grande, $L(A^n)$ non contiene alcuna stringa di lunghezza fissata, e nel limite otteniamo $X = AB$.*

Quando invece $\varepsilon \in L(A)$, l'equazione $X = AX + B$ può avere infinite soluzioni. In questo caso, la soluzione minimale è ancora $X = A^*B$, dove per "minimale" intendiamo il più piccolo linguaggio che soddisfa l'equazione. Tuttavia, anche linguaggi non regolari possono soddisfare l'equazione, quindi la situazione è più delicata.

Il lemma di Arden può essere generalizzato a sistemi di equazioni. Dato un automa a stati finiti con n stati, possiamo scrivere un sistema di n equazioni dove l'incognita X_i rappresenta l'insieme delle stringhe che permettono di raggiungere uno stato finale partendo dallo stato i . Risolvendo questo sistema mediante sostituzioni successive e applicazioni del lemma di Arden, otteniamo un'espressione regolare che descrive il linguaggio riconosciuto dall'automato.

Per esempio, consideriamo un automa con due stati q_0 (iniziale) e q_1 (finale), con transizioni $q_0 \xrightarrow{a} q_1$ e $q_1 \xrightarrow{b} q_1$. Il sistema di equazioni è: $X_0 = aX_1$, $X_1 = \varepsilon + bX_1$.

Dalla seconda equazione, applicando Arden con $A = b$ e $B = \varepsilon$, otteniamo $X_1 = b^*$. Sostituendo nella prima equazione, $X_0 = ab^*$. Quindi il linguaggio riconosciuto è descritto dall'espressione regolare ab^* .

Estensioni Algebriche: Intersezione e Complemento

Sebbene le espressioni regolari standard utilizzino solo unione, concatenazione e stella, è naturale chiedersi se l'aggiunta di altre operazioni aumenti la potenza espressiva. La risposta è negativa per molte operazioni naturali, ma la situazione è istruttiva dal punto di vista algebrico.

Il complemento di un linguaggio regolare è sempre regolare. Questo fatto non è ovvio dalla definizione di espressione regolare, ma segue immediatamente dalla caratterizzazione mediante automi: dato un automa deterministico completo per un linguaggio L , basta scambiare gli stati finali con quelli non finali per ottenere un automa per il complemento $\bar{L}$. Tuttavia, esprimere il complemento mediante espressioni regolari standard può richiedere un aumento esponenziale nella dimensione dell'espressione.

Analogamente, l'intersezione di due linguaggi regolari è regolare, ma non esiste una costruzione diretta di un'espressione regolare per $L_1 \cap L_2$ a partire dalle espressioni per L_1 e L_2 . La dimostrazione della regolarità dell'intersezione si basa sul prodotto di automi: dati

automi A_1 e A_2 per L_1 e L_2 , si costruisce un automa prodotto i cui stati sono coppie (q_1, q_2) di stati di A_1 e A_2 , e le transizioni sono definite componente per componente. Uno stato (q_1, q_2) è finale se e solo se entrambi q_1 e q_2 sono finali nei rispettivi automi.

Se estendiamo le espressioni regolari permettendo esplicitamente intersezione e complemento come operatori, otteniamo le cosiddette espressioni regolari estese. Queste hanno la stessa potenza espressiva delle espressioni regolari standard (descrivono esattamente i linguaggi regolari), ma possono essere esponenzialmente più concise. Per esempio, il linguaggio delle stringhe su $\{a, b\}$ che contengono un numero pari di a e un numero pari di b può essere descritto in modo naturale come intersezione di due linguaggi più semplici, ma richiede un'espressione regolare standard molto complessa.

La Struttura di Semianello dei Linguaggi Regolari

L'insieme $\mathcal{R}(\Sigma)$ dei linguaggi regolari su un alfabeto Σ , munito delle operazioni di unione e concatenazione, forma una struttura algebrica chiamata semianello. Un semianello è un insieme S dotato di due operazioni binarie $+$ e $\cdot$ che soddisfano:

$(S, +)$ è un monoide commutativo con elemento neutro 0 (nel nostro caso, l'unione con elemento neutro $\emptyset$).

$(S, \cdot)$ è un monoide con elemento neutro 1 (nel nostro caso, la concatenazione con elemento neutro $\{\epsilon\}$).

La moltiplicazione distribuisce rispetto all'addizione: $a(b + c) = ab + ac$ e $(a + b)c = ac + bc$.

L'elemento neutro dell'addizione è assorbente per la moltiplicazione: $0 \cdot a = a \cdot 0 = 0$.

Nel caso dei linguaggi regolari, abbiamo anche la stella di Kleene come operazione unaria. Un semianello dotato di un'operazione $*$ che soddisfa $1 + aa^* \leq a^*$ e $1 + aa \leq a$ (dove $\leq$ denota l'ordine indotto dall'unione, ovvero $L_1 \leq L_2$ se $L_1 \cup L_2 = L_2$) si chiama semianello di Kleene. L'insieme dei linguaggi regolari forma un semianello di Kleene libero, nel senso che ogni semianello di Kleene può essere ottenuto come immagine omomorfica dell'algebra delle espressioni regolari.

Questa prospettiva algebrica permette di studiare i linguaggi regolari usando strumenti della teoria dei semianelli. Per esempio, molti algoritmi per l'analisi di automi possono essere formulati come algoritmi di algebra lineare su semianelli, sostituendo le operazioni standard dell'addizione e moltiplicazione con unione e concatenazione di linguaggi.

Automi a Stati Finiti come Rappresentazioni Algebriche

Definizione Formale di Automa a Stati Finiti

Un automa a stati finiti (FSA, dall'inglese Finite State Automaton) è una struttura matematica che formalizza l'idea di una macchina con un numero finito di configurazioni interne (stati) che processa sequenze di simboli, cambiando stato in risposta a ciascun simbolo letto. Gli automi a stati finiti sono uno dei modelli di computazione più fondamentali e servono come

strumento teorico per comprendere le limitazioni e le possibilità dei sistemi computazionali più semplici.

Formalmente, un automa a stati finiti deterministico (DFA, Deterministic Finite Automaton) è una quintupla $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ dove:

Q è un insieme finito non vuoto di stati. Gli stati rappresentano le possibili configurazioni interne dell'automata. Benché Q possa essere qualsiasi insieme finito, spesso lo si rappresenta come $Q = \{q_0, q_1, \dots, q_{n-1}\}$ o con nomi più descrittivi.

Σ è l'alfabeto di input, un insieme finito di simboli che l'automata può leggere.

$\delta: Q \times \Sigma \rightarrow Q$ è la funzione di transizione, che specifica come l'automata cambia stato in risposta alla lettura di un simbolo. Dato uno stato corrente q e un simbolo a , $\delta(q, a)$ indica il nuovo stato in cui l'automata si trova dopo aver letto a mentre era nello stato q .

$q_0 \in Q$ è lo stato iniziale, lo stato in cui l'automata si trova prima di iniziare a leggere l'input.

$F \subseteq Q$ è l'insieme degli stati finali o accettanti. Una computazione ha successo se termina in uno di questi stati.

La funzione di transizione δ può essere estesa a stringhe intere definendo $\bar{\delta}: Q \times \Sigma^* \rightarrow Q$ ricorsivamente: $\bar{\delta}(q, \epsilon) = q$ per ogni stato q (leggere la stringa vuota non cambia lo stato). $\bar{\delta}(q, wa) = \bar{\delta}(\bar{\delta}(q, w), a)$ per ogni stato q , stringa w , e simbolo a .

In altre parole, per processare una stringa $w = a_1 a_2 \dots a_n$, l'automata parte dallo stato corrente q , legge a_1 e transita in $\bar{\delta}(q, a_1)$, poi legge a_2 e transita in $\bar{\delta}(\bar{\delta}(q, a_1), a_2)$, e così via. Il linguaggio riconosciuto o accettato dall'automata $\mathcal{A}$ è $L(\mathcal{A}) = \{w \in \Sigma^* : \bar{\delta}(q_0, w) \in F\}$, l'insieme delle stringhe che, partendo dallo stato iniziale, portano l'automata in uno stato finale.

Automi Non Deterministici

Un automa a stati finiti non deterministico (NFA, Nondeterministic Finite Automaton) generalizza il concetto di DFA permettendo che da uno stato, con un dato simbolo di input, l'automata possa transitare in più stati possibili. Formalmente, un NFA è una quintupla $\mathcal{N} = (Q, \Sigma, \Delta, q_0, F)$ dove Q, Σ, q_0, F hanno lo stesso significato che per i DFA, ma:

$\Delta: Q \times \Sigma \rightarrow \mathcal{P}(Q)$ è la funzione di transizione non deterministica, che associa a ogni coppia (q, a) un sottoinsieme di Q , ovvero l'insieme degli stati in cui l'automata può transitare.

L'estensione di Δ a stringhe si definisce come $\bar{\Delta}: Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$: $\bar{\Delta}(q, \epsilon) = \{q\}$. $\bar{\Delta}(q, wa) = \bigcup \{p \in \bar{\Delta}(q, w) : \bar{\Delta}(p, a)\}$.

Il linguaggio riconosciuto da $\mathcal{N}$ è $L(\mathcal{N}) = \{w \in \Sigma^* : \bar{\Delta}(q_0, w) \cap F \neq \emptyset\}$, ovvero l'insieme delle stringhe per cui esiste almeno un cammino computazionale che porta in uno stato finale.

Un NFA si dice ϵ -NFA se la funzione di transizione può includere anche transizioni spontanee, ovvero transizioni che l'automata può effettuare senza leggere alcun simbolo. Formalmente, la funzione di transizione diventa $\Delta: Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$. Le ϵ -transizioni

aumentano la flessibilità nella descrizione di automi, ma non incrementano la potenza computazionale.

Il teorema fondamentale sugli automi non deterministici afferma che per ogni NFA (o ε -NFA) esiste un DFA equivalente che riconosce lo stesso linguaggio. La dimostrazione si basa sulla costruzione per sottoinsiemi (powerset construction): gli stati del DFA equivalente sono i sottoinsiemi dell'insieme degli stati dell'NFA, e le transizioni sono definite applicando la funzione di transizione dell'NFA a tutti gli stati nel sottoinsieme. Questa costruzione può portare a un DFA con fino a 2^n stati se l'NFA ha n stati, e questo limite è stretto nel senso che esistono linguaggi per cui la conversione richiede effettivamente una crescita esponenziale.

Interpretazione Algebrica degli Automi

Gli automi a stati finiti ammettono un'elegante interpretazione algebrica che rivela profonde connessioni con la teoria dei monoidi. Dato un DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, possiamo associare a ogni stringa $w \in \Sigma^*$ una funzione $f_w: Q \rightarrow Q$ definita da $f_w(q) = \delta(q, w)$. Questa funzione descrive come la lettura di w trasforma gli stati dell'automa.

Le funzioni associate alle stringhe godono di importanti proprietà. Primo, la funzione associata alla stringa vuota è l'identità: $f_\epsilon(q) = q$ per ogni q . Secondo, la funzione associata alla concatenazione di due stringhe è la composizione delle funzioni associate: $f_{uv} = f_v \circ f_u$. Infatti, $f_{uv}(q) = \delta(q, uv) = \delta(\delta(q, u), v) = f_v(\delta(q, u)) = f_v(f_u(q)) = (f_v \circ f_u)(q)$.

L'insieme di tutte le funzioni $Q \rightarrow Q$ forma un monoide rispetto alla composizione, chiamato monoide di trasformazione su Q . L'applicazione $w \mapsto f_w$ definisce un morfismo di monoidi $h: \Sigma^* \rightarrow (Q \rightarrow Q)$, nel senso che h preserva la concatenazione: $h(uv) = h(u) \circ h(v)$. L'immagine di questo morfismo, ovvero l'insieme $M(\mathcal{A}) = \{f_w : w \in \Sigma^*\}$, è un sottomonoide finito del monoide di trasformazione, chiamato monoide di transizione dell'automa.

Un fatto notevole è che $M(\mathcal{A})$ è finito anche se Σ^* è infinito. Questo perché $M(\mathcal{A})$ è un sottoinsieme dell'insieme di tutte le funzioni $Q \rightarrow Q$, che ha cardinalità $|Q|^{|Q|}$, quindi finita. In realtà, $|M(\mathcal{A})| \leq |Q|^{|Q|}$, ma spesso $M(\mathcal{A})$ è molto più piccolo.

Il linguaggio riconosciuto dall'automa può essere caratterizzato algebricamente tramite il monoide di transizione. Diciamo che una funzione $f: Q \rightarrow Q$ è accettante se $f(q_0) \in F$. Allora $L(\mathcal{A}) = \{w \in \Sigma^* : f_w \text{ è accettante}\} = h^{-1}(M_{\text{acc}})$ dove $M_{\text{acc}} \subseteq M(\mathcal{A})$ è l'insieme delle funzioni accettanti. Questa caratterizzazione mostra che il linguaggio riconosciuto è esattamente l'immagine inversa di un sottoinsieme del monoide di transizione, collegando la teoria degli automi alla teoria dei morfismi di monoidi.

Il Teorema di Myhill-Nerode

Il teorema di Myhill-Nerode fornisce una caratterizzazione elegante dei linguaggi regolari in termini di congruenze di indice finito sul monoide libero Σ^* . Questo teorema è fondamentale perché collega la nozione sintattica di linguaggio regolare (esprimibile da un'espressione regolare o riconoscibile da un automa) con una caratterizzazione algebrica puramente in termini di relazioni di equivalenza.

Data un linguaggio $L \subseteq \Sigma^*$, definiamo la congruenza sintattica o congruenza di Myhill-Nerode di L come la relazione $\sim_L$ su Σ^* definita da:

$u \sim_L v$ se e solo se per ogni $x, y \in \Sigma^*$, $xuy \in L \Leftrightarrow xvy \in L$.

In altre parole, due stringhe u e v sono equivalenti modulo $\sim_L$ se sostituire u con v (o viceversa) in qualsiasi contesto non cambia l'appartenenza a L . È facile verificare che $\sim_L$ è effettivamente una congruenza: è riflessiva, simmetrica, transitiva, e compatibile con la concatenazione.

L'indice di $\sim_L$ è il numero di classi di equivalenza di $\sim_L$. Il teorema di Myhill-Nerode afferma che un linguaggio L è regolare se e solo se $\sim_L$ ha indice finito. Inoltre, se L è regolare, il numero minimo di stati di un DFA che riconosce L è esattamente l'indice di $\sim_L$.

La dimostrazione del teorema procede in due direzioni. Per la direzione "se", supponiamo che $\sim_L$ abbia indice finito n . Costruiamo un DFA i cui stati sono le classi di equivalenza $[w]$ di $\sim_L$. Lo stato iniziale è $[\epsilon]$, gli stati finali sono le classi $[w]$ tali che $w \in L$, e la funzione di transizione è $\delta([u], a) = [ua]$. Questa definizione è ben posta perché $\sim_L$ è una congruenza, quindi se $u \sim_L v$ allora $ua \sim_L va$. È facile verificare che questo automa riconosce esattamente L .

Per la direzione "solo se", supponiamo che L sia riconosciuto da un DFA con n stati. Definiamo una relazione $\sim_n$ su Σ^* ponendo $u \sim_n v$ se $\delta(q_0, u) = \delta(q_0, v)$, ovvero se u e v portano l'automata nello stesso stato partendo dallo stato iniziale. Questa relazione è una congruenza di indice al più n (perché ci sono al più n stati distinti). Inoltre, $\sim_n$ raffina $\sim_L$, nel senso che se $u \sim_n v$ allora $u \sim_L v$. Infatti, se u e v portano l'automata nello stesso stato q , allora per ogni stringa z , uz e vz portano l'automata in $\delta(q, z)$, quindi sono entrambe accettate o entrambe rifiutate. Poiché $\sim_n$ raffina $\sim_L$ e ha indice al più n , anche $\sim_L$ ha indice al più n .

Il teorema di Myhill-Nerode ha importanti conseguenze pratiche. Fornisce un criterio per dimostrare che un linguaggio non è regolare: basta mostrare che $\sim_L$ ha indice infinito. Per esempio, il linguaggio $L = \{a^n b^n : n \geq 0\}$ delle stringhe bilanciate di a e b non è regolare. Infatti, per ogni $n \neq m$, abbiamo $a^n \not\sim_L a^m$ perché $a^n b^n \in L$ ma $a^m b^n \notin L$. Quindi le stringhe $a, aa, aaa, \dots$ appartengono tutte a classi di equivalenza distinte di $\sim_L$, il che dimostra che $\sim_L$ ha indice infinito.

Minimizzazione di Automi

Il teorema di Myhill-Nerode implica che per ogni linguaggio regolare esiste un unico DFA minimo (a meno di isomorfismo) che lo riconosce, ovvero un DFA con il minor numero possibile di stati. Questo DFA minimo è esattamente l'automata quoziente definito dalle classi di equivalenza di $\sim_L$.

Data un DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ qualsiasi che riconosce L , possiamo costruire il DFA minimo mediante un algoritmo di minimizzazione. L'idea è di identificare e fondere stati che sono equivalenti dal punto di vista del linguaggio riconosciuto. Due stati p e q sono equivalenti se per ogni stringa w , $\delta(p, w) \in F$ se e solo se $\delta(q, w) \in F$, ovvero se portano ad accettazione esattamente per le stesse stringhe.

L'algoritmo classico di minimizzazione procede per raffinamenti successivi di una partizione degli stati. Inizialmente, partizioniamo Q in due classi: gli stati finali F e gli stati non finali $Q \setminus F$. Questi sono certamente non equivalenti perché raggiungibili con la stringa vuota portano a decisioni di accettazione diverse. Poi, raffiniamo iterativamente la partizione: due stati nella stessa classe vengono separati se esiste un simbolo a tale che con a transitano in classi diverse. Il processo termina quando non è più possibile raffinare la partizione.

Formalmente, definiamo una sequenza di equivalenze $\sim_0, \sim_1, \sim_2, \dots$ come segue: $p \sim_0 q$ se e solo se $(p \in F \Leftrightarrow q \in F)$. $p \sim_{k+1} q$ se e solo se $p \sim_k q$ e per ogni $a \in \Sigma$, $\delta(p, a) \sim_k \delta(q, a)$.

La sequenza delle equivalenze è decrescente: $\sim_{k+1} \subseteq \sim_k$ per ogni k . Poiché Q è finito, la sequenza deve stabilizzarsi dopo un numero finito di passi: esiste k_0 tale che $\sim_k = \sim_{k_0}$ per ogni $k \geq k_0$. L'equivalenza finale $\sim_{k_0}$ coincide con l'equivalenza che vogliamo, ovvero $p \sim_{k_0} q$ se e solo se per ogni w , $\delta(p, w) \in F \Leftrightarrow \delta(q, w) \in F$.

Una volta ottenuta la partizione finale, il DFA minimo ha come stati le classi di equivalenza, lo stato iniziale è la classe $[q_0]$, gli stati finali sono le classi che contengono almeno uno stato finale dell'automa originale, e la funzione di transizione è $\delta_{\min}([q], a) = [\delta(q, a)]$. Questa definizione è ben posta perché se $p \sim q$ allora $\delta(p, a) \sim \delta(q, a)$ per ogni a .

L'algoritmo di Hopcroft è un raffinamento che computa la partizione finale in tempo $O(n \log n)$ dove n è il numero di stati, migliorando l'algoritmo ingenuo che richiede tempo $O(n^2)$. L'idea chiave di Hopcroft è di processare le classi della partizione in un ordine intelligente per minimizzare il numero di raffinamenti necessari.

Un DFA si dice minimo se non esiste alcun DFA con meno stati che riconosce lo stesso linguaggio. Il DFA ottenuto dall'algoritmo di minimizzazione è minimo. Inoltre, tutti i DFA minimi per uno stesso linguaggio sono isomorfi, ovvero differiscono solo per una ridenominazione degli stati. Questo giustifica l'uso dell'articolo determinativo "il" DFA minimo.

Teoria dei Semigruppì Sintattici

Definizione e Proprietà Fondamentali

La teoria dei semigruppì sintattici rappresenta uno degli sviluppi più profondi nell'approccio algebrico ai linguaggi formali. Mentre il teorema di Myhill-Nerode caratterizza i linguaggi regolari tramite congruenze di indice finito, la teoria dei semigruppì va oltre e associa a ogni linguaggio regolare un oggetto algebrico canonico che cattura completamente le proprietà combinatorie del linguaggio.

Dato un linguaggio $L \subseteq \Sigma^*$, la congruenza sintattica $\sim_L$ induce un monoide quoziente $\Sigma^*/\sim_L$. Tuttavia, è spesso più conveniente lavorare con il semigruppì sintattico, che è definito a partire dal sottoinsieme Σ^+ delle stringhe non vuote. La congruenza sintattica restringe a una congruenza su Σ^+ , e il quoziente $\Sigma^*/\sim_L$ è un semigruppì (un monoide senza necessariamente l'elemento neutro).

Il semigruppato sintattico di L , denotato $\text{Synt}(L)$, è il più piccolo semigruppato (o monoide, se si include la stringa vuota) che riconosce L nel senso seguente: esiste un morfismo $\varphi: \Sigma^+ \rightarrow \text{Synt}(L)$ tale che L è l'immagine inversa di un sottoinsieme di $\text{Synt}(L)$, ovvero $L = \varphi^{-1}(P)$ per qualche $P \subseteq \text{Synt}(L)$. Il morfismo φ è definito come la proiezione canonica $\varphi(w) = [w]$, la classe di equivalenza di w rispetto a $\sim_L$.

Una proprietà fondamentale del semigruppato sintattico è che divide ogni altro semigruppato che riconosce L , nel senso della teoria dei semigruppato. Dire che un semigruppato S divide un semigruppato T significa che S è un quoziente di un sottosemigruppato di T . Questa proprietà di minimalità rende il semigruppato sintattico un invariante canonico del linguaggio.

Il semigruppato sintattico è finito se e solo se il linguaggio è regolare. Questa è essenzialmente una riformulazione del teorema di Myhill-Nerode in linguaggio semigruppato. La dimensione del semigruppato sintattico (il suo ordine) è esattamente l'indice della congruenza sintattica, e quindi coincide con il numero di stati del DFA minimo (eventualmente +1 se si considera il monoide invece del semigruppato).

Classificazione dei Linguaggi Regolari tramite Semigruppato

Una delle applicazioni più potenti della teoria dei semigruppato sintattici è la classificazione dei linguaggi regolari in base alle proprietà algebriche dei loro semigruppato. Linguaggi con proprietà strutturali simili tendono ad avere semigruppato sintattici con proprietà algebriche simili, e viceversa.

Per esempio, un linguaggio L è riconosciuto da un automa senza cicli (un automa aciclico) se e solo se il suo semigruppato sintattico è un semigruppato aperiodico, ovvero un semigruppato in cui ogni elemento ha un'unica potenza idempotente. Un elemento s di un semigruppato è idempotente se $s^2 = s$. In ogni semigruppato finito, ogni elemento s ha almeno una potenza idempotente, perché la sequenza $s, s^2, s^3, \dots$ deve eventualmente ripetersi, e il primo elemento ripetuto in questa sequenza è idempotente. Un semigruppato è aperiodico se per ogni elemento s esiste un intero n tale che $s^n = s^{n+1}$.

Il teorema di Schützenberger afferma che un linguaggio è star-free (ovvero può essere descritto da un'espressione regolare senza usare la stella di Kleene, ma usando solo unione, concatenazione e complemento) se e solo se il suo semigruppato sintattico è aperiodico. Questo risultato crea una connessione sorprendente tra una restrizione sintattica sulle espressioni regolari e una proprietà algebrica del semigruppato sintattico.

Per esempio, il linguaggio $L = \{w \in \{a, b\}^* : w \text{ contiene } aa\}$ è star-free perché può essere descritto dall'espressione $(a \cup b)aa(a \cup b)$, ma anche come complemento di $((b)^*a(b)^*a(b)^*)$, e utilizzando le leggi di De Morgan si può riscrivere senza stella. Il semigruppato sintattico di L ha quattro elementi corrispondenti alle classi: stringhe che non contengono a , stringhe che terminano con esattamente una a , stringhe che contengono aa , e la classe vuota.

Un altro esempio importante è la gerarchia di Straubing-Thérien, che classifica i linguaggi regolari in base alla complessità dei loro semigruppato sintattici. Questa gerarchia parte dai linguaggi la cui appartenenza dipende solo dal simbolo iniziale e finale (livello 0), passa attraverso i linguaggi definibili in logica del primo ordine con predicati di ordine (livello 1), e

continua con livelli sempre più complessi. Ogni livello corrisponde a una classe di semigrupperi definita algebricamente, creando una corrispondenza profonda tra logica, algebra e teoria dei linguaggi.

Semigrupperi e Automi: Il Teorema di Krohn-Rhodes

Il teorema di decomposizione di Krohn-Rhodes è uno dei risultati più profondi nella teoria algebrica degli automi. Esso afferma che ogni semigruppero finito può essere decomposto come un wreath product (prodotto ghirlanda) di semigrupperi più semplici, in particolare gruppi semplici e semigrupperi a tre elementi chiamati semigrupperi flip-flop.

Dal punto di vista degli automi, questo significa che ogni automa può essere simulato da una cascata di automi più semplici, ciascuno dei quali è o un automa di gruppo (un automa il cui semigruppero di transizione è un gruppo) o un automa reset (un automa con capacità di "reset" dello stato). Questa decomposizione è analoga alla fattorizzazione dei numeri in primi: così come ogni intero può essere scritto come prodotto di numeri primi, ogni semigruppero finito può essere decomposto in semigrupperi "irriducibili".

La complessità di un semigruppero S , denotata con $c(S)$, è la lunghezza minima di una decomposizione di S in componenti semplici. Il teorema di Krohn-Rhodes stabilisce che $c(S)$ è ben definito e finito per ogni semigruppero finito S . Per i semigrupperi aperiodici, $c(S)$ è sempre al più il numero di stati meno uno, fornendo un limite superiore stretto sulla complessità.

Questo teorema ha profonde implicazioni per la teoria della complessità computazionale e per la progettazione di circuiti. Mostra che sistemi complessi possono essere sempre decomposti in componenti semplici che interagiscono in modo gerarchico, un principio che si ritrova in molti ambiti dell'informatica e dell'ingegneria.

Varietà di Linguaggi e Pseudovarietà di Semigrupperi

Eilenberg stabilì negli anni Settanta una corrispondenza fondamentale tra varietà di linguaggi e pseudovarietà di semigrupperi finiti. Una varietà di linguaggi è una famiglia di linguaggi chiusa rispetto a certe operazioni naturali (immagini inverse tramite morfismi, intersezioni con linguaggi regolari, ecc.). Una pseudovarietà di semigrupperi è una classe di semigrupperi finiti chiusa rispetto a quozienti, sottosemigrupperi e prodotti diretti finiti.

Il teorema di Eilenberg afferma che esiste una corrispondenza biunivoca tra varietà di linguaggi regolari e pseudovarietà di semigrupperi finiti: a ogni varietà V di linguaggi corrisponde una pseudovarietà W di semigrupperi tale che un linguaggio L appartiene a V se e solo se il suo semigruppero sintattico appartiene a W .

Per esempio, la varietà dei linguaggi star-free corrisponde alla pseudovarietà dei semigrupperi aperiodici. La varietà dei linguaggi riconoscibili da automi contatori modulari (automi che contano modulo n) corrisponde alla pseudovarietà dei gruppi abeliani. Queste corrispondenze permettono di tradurre problemi sui linguaggi in problemi sui semigrupperi e viceversa.

Un'applicazione importante è la decidibilità di appartenenza a varietà. Se possiamo caratterizzare algebricamente una pseudovarietà (per esempio mediante identità equazionali), possiamo spesso decidere algebricamente se un dato linguaggio regolare appartiene alla varietà corrispondente, semplicemente calcolando il suo semigrupp sintattico e verificando se soddisfa le identità richieste.

Il Monoide delle Trasformazioni e Rappresentazioni

Ogni automa a stati finiti induce naturalmente una rappresentazione del monoide libero Σ^* nel monoide delle trasformazioni sullo spazio degli stati. Questa rappresentazione cattura completamente il comportamento dell'automato e permette di studiarlo usando strumenti dell'algebra lineare (generalizzata ai semianelli).

Dato un automa $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ con $n = |Q|$ stati, possiamo rappresentare ogni trasformazione $f_a: Q \rightarrow Q$ associata a una stringa w come una matrice $n \times n$ su un semianello appropriato. Se usiamo il semianello booleano $\{0, 1\}$, la matrice M_a ha entrata (i, j) uguale a 1 se f_a porta lo stato i nello stato j , e 0 altrimenti.

La concatenazione di stringhe corrisponde al prodotto di matrici: $M_{\{uv\}} = M_u \cdot M_v$. Il riconoscimento di una stringa w da parte dell'automato può essere verificato calcolando M_w e controllando se la riga corrispondente allo stato iniziale ha un 1 in una colonna corrispondente a uno stato finale.

Questa rappresentazione matriciale permette di utilizzare algoritmi di algebra lineare per analizzare automi. Per esempio, il problema della raggiungibilità (determinare se esiste una stringa che porta da uno stato p a uno stato q) può essere risolto calcolando la chiusura transitiva della matrice di adiacenza dell'automato, che rappresenta le transizioni con singoli simboli.

Inoltre, la rappresentazione matriciale si estende naturalmente a semianelli più generali. Se usiamo il semianello dei numeri naturali $\mathbb{N}$, le entrate della matrice M_w contano il numero di modi in cui la stringa w può portare da uno stato all'altro (rilevante per automi non deterministici). Se usiamo il semianello tropicale, possiamo risolvere problemi di cammino minimo negli automi pesati.

Pumping Lemmas e Dimostrazioni Algebriche

Il Pumping Lemma Classico per Linguaggi Regolari

Il pumping lemma è uno strumento fondamentale per dimostrare che certi linguaggi non sono regolari. La formulazione classica afferma che se L è un linguaggio regolare, allora esiste una costante p (chiamata lunghezza di pompaggio) tale che ogni stringa w in L con $|w| \geq p$ può essere decomposta come $w = xyz$ dove:

1. $|xy| \leq p$
2. $|y| \geq 1$
3. Per ogni $i \geq 0$, la stringa xy^iz appartiene a L

La dimostrazione standard del pumping lemma si basa sul principio dei cassetti (pigeonhole principle). Se L è riconosciuto da un DFA con n stati, prendiamo $p = n$ come lunghezza di pompaggio. Consideriamo una stringa $w = a_1 a_2 \dots a_m$ con $m \geq n$. La computazione dell'automa su w visita una sequenza di stati $q_0, q_1, q_2, \dots, q_m$ dove $q_i = \delta^*(q_0, a_1 \dots a_i)$. Poiché $m \geq n$ e ci sono solo n stati disponibili, per il principio dei cassetti almeno due degli stati $q_0, q_1, \dots, q_m$ devono coincidere. Sia $q_i = q_j$ con $i < j \leq n$.

Allora possiamo decomporre w come xyz dove $x = a_1 \dots a_i$, $y = a_{i+1} \dots a_j$, e $z = a_{j+1} \dots a_m$. Poiché $q_i = q_j$, la lettura di y forma un ciclo nello spazio degli stati: partendo da q_i e leggendo y si ritorna a q_i . Questo significa che possiamo ripetere y quante volte vogliamo (ottenendo $xy^i z$ per $i \geq 0$) o ometterla completamente (ottenendo $xz = xy^0 z$), e in tutti i casi l'automa transita attraverso la stessa sequenza di stati $q_0, \dots, q_i, \dots, q_j$ (dove il ciclo y è ripetuto i volte o omissso). Se w è accettato, anche tutte le stringhe $xy^i z$ sono accettate.

Formulazione Algebrica del Pumping Lemma

Il pumping lemma ammette una formulazione elegante in termini di semigrupp sintattici che rivela la sua natura algebrica profonda. Se L è un linguaggio regolare con semigrupp sintattico S di cardinalità n , allora per ogni stringa w con $|w| \geq n$, esistono stringhe u, v, z tali che $w = uvz$ e v corrisponde a un elemento idempotente di S .

Più precisamente, sia $\varphi: \Sigma^* \rightarrow S$ il morfismo canonico. Se $|w| \geq n$ e $w = a_1 a_2 \dots a_m$, consideriamo gli elementi $\varphi(a_1), \varphi(a_1 a_2), \dots, \varphi(a_1 \dots a_m)$. Poiché S ha cardinalità n , almeno due di questi elementi devono coincidere: $\varphi(a_1 \dots a_i) = \varphi(a_1 \dots a_j)$ per qualche $i < j \leq n$. Posto $u = a_1 \dots a_i$, $v = a_{i+1} \dots a_j$, abbiamo $\varphi(u) \cdot \varphi(v) = \varphi(uv) = \varphi(a_1 \dots a_j) = \varphi(a_1 \dots a_i) = \varphi(u)$.

Quindi $\varphi(v)$ è un elemento e tale che $\varphi(u) \cdot e = \varphi(u)$. Considerando potenze di v , abbiamo $\varphi(v^k) = \varphi(v)^k$. In un semigrupp finito, per ogni elemento s esiste un intero m tale che s^m è idempotente ($s^m \cdot s^m = s^m$). Quindi esiste k tale che $\varphi(v^k)$ è idempotente.

Questa formulazione algebrica fa vedere che il pumping non è un accidente della rappresentazione mediante automi, ma riflette una proprietà intrinseca della struttura algebrica del linguaggio. L'esistenza di elementi idempotenti nel semigrupp sintattico corrisponde all'esistenza di cicli negli automi, ma la caratterizzazione algebrica è più astratta e non dipende dalla particolare rappresentazione scelta.

Applicazioni del Pumping Lemma

Il pumping lemma è utilizzato principalmente per dimostrare la non regolarità di linguaggi. La tecnica standard è per contraddizione: si assume che il linguaggio sia regolare, quindi deve esistere una lunghezza di pompaggio p ; si sceglie poi una stringa specifica w in L con $|w| \geq p$; si dimostra che per ogni decomposizione $w = xyz$ soddisfacente le condizioni del lemma, esiste un i tale che $xy^i z$ non appartiene a L , contraddicendo il lemma.

Esempio classico: il linguaggio $L = \{a^n b^n : n \geq 0\}$ non è regolare. Supponiamo per assurdo che L sia regolare con lunghezza di pompaggio p . Consideriamo la stringa $w = a^p b^p$. Per il pumping lemma, $w = xyz$ con $|xy| \leq p$, $|y| \geq 1$, e $xy^i z \in L$ per ogni $i \geq 0$. Poiché $|xy| \leq p$, la sottostringa xy è contenuta interamente nella parte a^p di w , quindi $y = a^k$ per qualche $k \geq 1$. Allora $xy^2 z = a^{p+k} b^p$, che non appartiene a L perché ha più a che b . Contraddizione.

Esempio più sofisticato: il linguaggio $L = \{a^{n^2} : n \geq 0\}$ delle stringhe la cui lunghezza è un quadrato perfetto non è regolare. Supponiamo che sia regolare con lunghezza di pompaggio p . Prendiamo $w = a^{p^2}$. Allora $w = xyz$ con $|y| = k$ per qualche $1 \leq k \leq p$. Consideriamo $xy^2z = a^{p^2+k}$. Per appartenere a L , dovremmo avere $p^2 + k = m^2$ per qualche intero m . Ma $p^2 < p^2 + k \leq p^2 + p < p^2 + 2p + 1 = (p + 1)^2$, quindi non esiste alcun quadrato perfetto tra p^2 e $(p + 1)^2$ se $k > 0$. Contraddizione.

Il Pumping Lemma per Linguaggi Context-Free

Esiste anche un pumping lemma per i linguaggi context-free, più complesso di quello per linguaggi regolari. Esso afferma che se L è context-free, esiste una costante p tale che ogni stringa w in L con $|w| \geq p$ può essere scritta come $w = uvxyz$ dove:

1. $|vxy| \leq p$
2. $|vy| \geq 1$
3. Per ogni $i \geq 0$, $uv^ixy^iz \in L$

La differenza cruciale rispetto al pumping lemma per linguaggi regolari è che qui pompiamo due sottostringhe v e y simultaneamente, e queste possono trovarsi in posizioni distanti nella stringa. Questa differenza riflette la maggiore potenza espressiva delle grammatiche context-free rispetto agli automi a stati finiti.

La dimostrazione si basa sull'albero di derivazione. Se L è generato da una grammatica context-free con k variabili e lunghezza massima delle parti destre delle regole pari a b , consideriamo una stringa w di lunghezza almeno b^{k+1} . L'albero di derivazione di w ha altezza almeno $k + 1$. Per il principio dei cassetti, lungo un cammino dalla radice a una foglia deve esistere una variabile A che appare due volte. Questo crea una situazione analoga al ciclo negli automi: possiamo ripetere la derivazione dalla prima occorrenza di A alla seconda, ottenendo stringhe uv^ixy^iz .

Un esempio di linguaggio non context-free è $L = \{a^n b^n c^n : n \geq 0\}$. Supponiamo che L sia context-free con costante p . Consideriamo $w = a^p b^p c^p$. Per il pumping lemma, $w = uvxyz$ con $|vxy| \leq p$, $|vy| \geq 1$, e $uv^ixy^iz \in L$ per ogni i . Poiché $|vxy| \leq p$, vxy può contenere al massimo due dei tre tipi di simboli (a , b , c). Quindi pompando ($i = 2$) aumentiamo il conteggio di al massimo due tipi di simboli, mentre il terzo rimane costante. Questo rompe l'equilibrio $a^n b^n c^n$, contraddicendo l'appartenenza a L .

Versioni Rafforzate: Il Fine and Wilf Theorem

Il teorema di Fine e Wilf è un risultato combinatorio profondo sulle stringhe che può essere visto come una versione raffinata del concetto di periodicità. Esso afferma che se una stringa w ha periodi p e q , e se $|w| \geq p + q - \gcd(p, q)$, allora w ha anche periodo $\gcd(p, q)$.

Questo teorema ha applicazioni importanti nella dimostrazione di proprietà di linguaggi. Per esempio, può essere usato per caratterizzare i linguaggi che soddisfano certe proprietà di periodicità. Un linguaggio L si dice periodico con periodo k se per ogni stringa $w \in L$ e ogni posizione $i \geq k$, anche la stringa w' ottenuta da w spostando circolarmente i simboli di k posizioni appartiene a L .

La dimostrazione del teorema di Fine e Wilf è costruttiva e utilizza l'algoritmo euclideo. Se w ha periodi p e q , possiamo "allineare" le due strutture periodiche e mostrare che coincidono su intervalli sempre più piccoli, fino a raggiungere un periodo comune uguale a $\gcd(p, q)$. L'ipotesi sulla lunghezza è necessaria: esistono stringhe di lunghezza $p + q - \gcd(p, q) - 1$ che hanno periodi p e q ma non $\gcd(p, q)$.

Un'applicazione algebrica del teorema di Fine e Wilf è la caratterizzazione dei linguaggi commutanti. Due linguaggi L_1 e L_2 commutano se $L_1 L_2 = L_2 L_1$. Caratterizzare quando due linguaggi regolari commutano è un problema non banale che coinvolge analisi profonde della struttura periodica delle stringhe nei linguaggi. Il teorema di Fine e Wilf fornisce strumenti essenziali per affrontare questo problema.

Minimizzazione di Automi: Prospettiva Algebrica Completa

L'Algoritmo di Moore e Raffinamenti Successivi

L'algoritmo di Moore per la minimizzazione di automi rappresenta una delle prime applicazioni sistematiche dell'idea di raffinamento di partizioni in informatica. L'algoritmo parte dalla partizione più grossolana compatibile con il riconoscimento (stati finali vs non finali) e la raffina iterativamente fino a raggiungere la partizione più fine possibile, corrispondente alla relazione di equivalenza degli stati.

Formalmente, dato un DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, definiamo una sequenza di partizioni $P_0, P_1, P_2, \dots$ di Q come segue:

$P_0 = \{F, Q \setminus F\}$, la partizione iniziale che separa stati finali da non finali.

P_{i+1} è il raffinamento di P_i ottenuto separando due stati p, q nella stessa classe di P_i se esiste un simbolo $a \in \Sigma$ tale che $\delta(p, a)$ e $\delta(q, a)$ appartengono a classi diverse in P_i .

La sequenza di partizioni è strettamente decrescente nel senso che ogni partizione è un raffinamento della precedente: $P_{i+1} \subseteq P_i$ (nel senso dell'ordine sulle partizioni, dove $P \subseteq P'$ significa che ogni classe di P è contenuta in qualche classe di P'). Poiché Q è finito, la sequenza deve stabilizzarsi dopo un numero finito di passi. Sia P^* la partizione limite.

La partizione P^* caratterizza esattamente l'equivalenza di stati: due stati p e q appartengono alla stessa classe di P^* se e solo se sono equivalenti nel senso che $\delta(p, w) \in F \Leftrightarrow \delta(q, w) \in F$ per ogni $w \in \Sigma^*$. Questo si dimostra per induzione: se p e q sono separati al passo k , allora esiste una stringa di lunghezza al più k che li distingue; viceversa, se p e q sono distinti al passo k , allora sono distinguibili da qualche stringa.

L'automa minimo è costruito prendendo le classi di P^* come stati, la classe contenente q_0 come stato iniziale, le classi che intersecano F come stati finali, e definendo $\delta_{\min}([p], a) = [\delta(p, a)]$. Questa costruzione è ben posta perché se p e q appartengono alla stessa classe, allora $\delta(p, a)$ e $\delta(q, a)$ appartengono alla stessa classe per definizione di P^* .

La complessità temporale dell'algoritmo di Moore nel caso peggiore è $O(n^2|\Sigma|)$ dove $n = |Q|$, perché possono essere necessari fino a $n - 1$ raffinamenti, e ogni raffinamento richiede tempo $O(n|\Sigma|)$ per controllare tutte le transizioni. Questo può essere migliorato a $O(n \log n |\Sigma|)$ con l'algoritmo di Hopcroft.

L'Algoritmo di Hopcroft e Complessità Ottimale

L'algoritmo di Hopcroft rappresenta un raffinamento sofisticato dell'idea di Moore, ottimizzando l'ordine in cui le classi vengono processate per minimizzare il numero di operazioni. L'idea chiave è di mantenere un insieme di "classi distrettrici" (splitting classes) e processarle in un ordine intelligente.

L'algoritmo mantiene una partizione corrente P e un insieme di lavoro W contenente classi che devono ancora essere usate per raffinare P . Inizialmente, $P = \{F, Q \setminus F\}$ e W contiene la classe più piccola tra F e $Q \setminus F$. A ogni passo, si estrae una classe C da W e un simbolo a , e si raffina P rispetto alla pre-immagine di C tramite a , ovvero si separa ogni classe B di P in due parti: gli stati da cui con a si raggiunge C , e gli stati da cui con a non si raggiunge C .

La scelta cruciale di Hopcroft è di aggiungere a W sempre la classe più piccola risultante da una separazione. Questo garantisce che ogni stato viene "processato" al più $O(\log n)$ volte, portando alla complessità totale $O(n \log n |\Sigma|)$.

La dimostrazione della correttezza è simile a quella di Moore: l'algoritmo termina quando W è vuoto, il che significa che non ci sono più raffinamenti possibili. La dimostrazione della complessità richiede un'analisi ammortizzata sofisticata: si associa a ogni stato un "potenziale" proporzionale al logaritmo della dimensione della sua classe corrente, e si mostra che il potenziale totale può aumentare al più $O(n \log n)$ volte.

L'algoritmo di Hopcroft è ottimale nel modello di computazione basato sul confronto di stati, nel senso che esistono famiglie di automi per cui ogni algoritmo di minimizzazione basato su raffinamenti di partizioni richiede tempo $\Omega(n \log n)$. Questo è stato dimostrato da Hopcroft stesso nel suo articolo originale del 1971.

Minimizzazione di NFA e Limiti Intrinseci

A differenza dei DFA, la minimizzazione di automi non deterministici (NFA) è un problema molto più difficile. Il problema di determinare se esiste un NFA con al più k stati equivalente a un dato NFA è PSPACE-completo, il che suggerisce che non esiste un algoritmo efficiente generale per la minimizzazione di NFA.

La difficoltà risiede nel fatto che la nozione di "stato equivalente" per NFA è più sottile. In un DFA, due stati sono equivalenti se accettano esattamente le stesse stringhe. In un NFA, dobbiamo considerare tutte le possibili computazioni non deterministiche, e la struttura delle transizioni diventa cruciale, non solo il linguaggio accettato da ciascuno stato.

Esistono tuttavia risultati parziali. Per esempio, se restringiamo l'attenzione a NFA senza ϵ -transizioni, il problema della minimizzazione diventa più trattabile, anche se rimane NP-hard in generale. Esistono algoritmi euristici che funzionano bene in pratica, basati su simulazioni e bisimulazioni tra stati.

Un risultato notevole è che la conversione da NFA a DFA minimo può richiedere una crescita esponenziale nel numero di stati: esistono linguaggi riconosciuti da NFA con n stati il cui DFA minimo ha 2^n stati. Questo dimostra che il non determinismo può essere esponenzialmente più conciso del determinismo per certi linguaggi.

Minimizzazione Modulo Bisimulazione

Un approccio alternativo alla minimizzazione si basa sulla nozione di bisimulazione piuttosto che su equivalenza di linguaggi. Due stati p e q sono bisimili se esiste una relazione di bisimulazione che li collega, dove una bisimulazione è una relazione R tale che se $(p, q) \in R$, allora:

- $p \in F$ se e solo se $q \in F$
- Per ogni simbolo a e stato p' tale che $\delta(p, a) = p'$, esiste q' tale che $\delta(q, a) = q'$ e $(p', q') \in R$
- Simmetricamente per transizioni da q

La bisimulazione cattura l'idea che due stati hanno "lo stesso comportamento" non solo in termini di stringhe accettate, ma anche in termini di struttura delle transizioni. Per DFA, bisimulazione coincide con equivalenza di linguaggi, ma per NFA o sistemi di transizione più generali, la bisimulazione è una relazione più fine.

L'algoritmo di Paige-Tarjan calcola la massima bisimulazione (e quindi l'automa quoziente rispetto alla bisimulazione) in tempo $O(m \log n)$ dove m è il numero di transizioni e n il numero di stati. Questo algoritmo è basato su tecniche di raffinamento di partizioni simili a Hopcroft, ma adattate al contesto della bisimulazione.

La minimizzazione rispetto alla bisimulazione è importante in model checking e verifica formale, dove la struttura delle transizioni è rilevante tanto quanto il linguaggio riconosciuto.

Complessità Descrittiva e Caratterizzazioni Logiche

Logiche e Linguaggi Regolari

La teoria della complessità descrittiva studia la potenza espressiva di vari formalismi logici per descrivere linguaggi e proprietà computazionali. Per i linguaggi regolari, esistono caratterizzazioni logiche profonde che rivelano connessioni sorprendenti tra logica matematica e teoria degli automi.

Un formalismo logico fondamentale è la logica monadica del second'ordine (MSO, Monadic Second-Order logic) sulle stringhe. Le formule MSO possono quantificare su posizioni individuali nelle stringhe (variabili del prim'ordine) e su insiemi di posizioni (variabili del second'ordine monadiche). I predicati atomici includono: "la posizione x è etichettata con il simbolo a ", " $x < y$ " (ordine tra posizioni), e " $x \in X$ " (appartenenza a un insieme).

Il teorema di Büchi-Elgot-Trakhtenbrot afferma che un linguaggio è regolare se e solo se è definibile in MSO. Più precisamente, un linguaggio $L \subseteq \Sigma^*$ è regolare se e solo se esiste una formula MSO φ tale che $w \in L$ se e solo se la struttura di stringa associata a w soddisfa φ .

Per esempio, il linguaggio delle stringhe su $\{a, b\}$ che contengono un numero pari di a può essere espresso dalla formula MSO: $\exists X. (\forall x. x \in X \leftrightarrow [a(x) \wedge \neg \exists Y. (x \in Y \wedge \forall z < x. (a(z) \rightarrow z \in Y) \wedge |Y| \text{ è dispari})])$

Questa caratterizzazione stabilisce un ponte profondo tra logica e automazione. Da un lato, ogni automa può essere tradotto in una formula MSO che descrive il linguaggio riconosciuto. Dall'altro, ogni formula MSO può essere compilata in un automa equivalente, usando tecniche di eliminazione dei quantificatori.

Logiche del Prim'Ordine e Linguaggi Star-Free

Una restrizione importante di MSO è la logica del prim'ordine (FO, First-Order logic), dove sono permessi solo quantificatori su posizioni individuali, non su insiemi. La logica FO è strettamente meno espressiva di MSO: non può esprimere tutte le proprietà regolari.

Il teorema di McNaughton-Papert caratterizza esattamente i linguaggi regolari definibili in FO: sono precisamente i linguaggi star-free, ovvero quelli descrivibili da espressioni regolari senza usare la stella di Kleene (ma usando unione, concatenazione e complemento). Questo risultato collega logica, espressioni regolari e algebra: i linguaggi FO-definibili sono quelli con semigrupp sintattico aperiodico.

Per esempio, il linguaggio $L = \text{"le stringhe su } \{a, b\} \text{ che contengono la sottostringa } ab\text{"}$ è star-free ed è esprimibile in FO come: $\exists x \exists y. (x < y \wedge a(x) \wedge b(y) \wedge \forall z. (x < z < y \rightarrow \perp))$

Il linguaggio $L' = \text{"le stringhe su } \{a\} \text{ la cui lunghezza è pari}"$ invece non è star-free (ha semigrupp sintattico ciclico $Z/2Z$, che non è aperiodico) e non è definibile in FO, anche se è ovviamente regolare.

Gerarchie Logiche e Classificazione di Linguaggi

Esistono molte gerarchie logiche intermedie tra FO e MSO che corrispondono a classi naturali di linguaggi regolari. La gerarchia di quantificatori alternati, per esempio, classifica le formule FO in base al numero di alternanze tra quantificatori esistenziali e universali. Il livello Σ_n della gerarchia consiste delle formule che iniziano con n blocchi di quantificatori, alternando tra esistenziali e universali, mentre il livello Π_n inizia con quantificatori universali.

È notevole che ogni livello di questa gerarchia corrisponde a una classe specifica di linguaggi regolari con caratterizzazione algebrica. Il livello Σ_1 , contenente formule della forma $\exists x_1 \dots \exists x_n \varphi$ dove φ è senza quantificatori, corrisponde esattamente ai linguaggi i cui semigrupp sintattici soddisfano certe identità algebriche specifiche. Questi linguaggi sono riconoscibili da automi con particolari restrizioni sulla struttura delle transizioni.

La dimostrazione di queste corrispondenze utilizza tecniche sofisticate di teoria dei modelli e algebra universale. L'idea centrale è che i quantificatori logici corrispondono a operazioni algebriche sui semigrupp: il quantificatore esistenziale corrisponde a una forma di join o unione, mentre il quantificatore universale corrisponde a una forma di meet o intersezione. Le alternanze di quantificatori si traducono in composizioni di queste operazioni algebriche, creando una gerarchia naturale sia dal lato logico che da quello algebrico.

Un risultato profondo dovuto a Thomas afferma che la gerarchia di quantificatori su stringhe è strettamente infinita: per ogni n , esistono linguaggi definibili in Σ_{n+1} che non sono definibili in Σ_n . Questo contrasta con la situazione per strutture finite generali, dove la gerarchia collassa al livello finito per risultati di Fagin. La differenza cruciale è che le stringhe hanno una struttura d'ordine lineare ricca che permette di codificare informazioni arbitrariamente complesse attraverso quantificatori annidati.

Logiche Temporal e Automi su Parole Infinite

Sebbene il nostro focus principale sia sui linguaggi di stringhe finite, vale la pena menzionare brevemente l'estensione ai linguaggi di stringhe infinite, o ω -linguaggi. Questi sono fondamentali nella verifica di sistemi reattivi, dove si modellano comportamenti che continuano indefinitamente.

La logica temporale lineare (LTL, Linear Temporal Logic) è un formalismo per esprimere proprietà su sequenze infinite. LTL estende la logica proposizionale con operatori temporali come "next" (X), "eventually" (F), "globally" (G), e "until" (U). Per esempio, la formula $G(\text{request} \rightarrow F \text{ grant})$ esprime che ogni richiesta è eventualmente seguita da una concessione.

Il teorema fondamentale di Büchi afferma che un ω -linguaggio è riconoscibile da un automa di Büchi (un automa su stringhe infinite con una condizione di accettazione modificata) se e solo se è definibile in MSO sugli ω -ordini. Inoltre, Kamp dimostrò che LTL ha la stessa potenza espressiva di FO sui ω -ordini, creando una corrispondenza simile a quella di McNaughton-Papert per stringhe finite.

La teoria algebrica degli ω -linguaggi è più sottile di quella per linguaggi finiti. I semigrupp vengono sostituiti da ω -semigrupp, strutture che incorporano sia moltiplicazione finita che infinita. La classificazione degli ω -linguaggi regolari tramite ω -semigrupp sintattici, sviluppata da Perrin e Pin, rivela una ricchezza strutturale ancora maggiore rispetto al caso finito.

Proprietà di Chiusura e Operazioni sui Linguaggi

Chiusura dei Linguaggi Regolari

Una delle caratteristiche più importanti della classe dei linguaggi regolari è la sua robustezza rispetto a molte operazioni naturali. Questa robustezza si manifesta attraverso numerose proprietà di chiusura che dimostrano come i linguaggi regolari formino una classe algebricamente ben comportata.

I linguaggi regolari sono chiusi rispetto all'unione. Se L_1 e L_2 sono regolari, allora $L_1 \cup L_2$ è regolare. Questo è immediato dalla definizione di espressione regolare: se r_1 denota L_1 e r_2 denota L_2 , allora $r_1 + r_2$ denota $L_1 \cup L_2$. Dal punto di vista degli automi, dati DFA A_1 e A_2 per L_1 e L_2 , possiamo costruire un NFA per $L_1 \cup L_2$ introducendo un nuovo stato iniziale con ϵ -transizioni agli stati iniziali di A_1 e A_2 , e prendendo come stati finali l'unione degli stati finali di A_1 e A_2 .

La concatenazione preserva anch'essa la regolarità. Se L_1 e L_2 sono regolari, lo è anche L_1L_2 . Questo segue dalla definizione di espressione regolare: r_1r_2 denota L_1L_2 . La costruzione mediante automi è leggermente più delicata: dato un NFA per L_1 con stati finali F_1 e un NFA per L_2 con stato iniziale q_0^2 , aggiungiamo ε -transizioni da ogni stato in F_1 a q_0^2 , creando così un automa per la concatenazione.

La stella di Kleene, come già discusso, è parte integrante della definizione di linguaggi regolari, quindi la chiusura rispetto a questa operazione è automatica. Se L è regolare, L^* è regolare. La costruzione mediante automi introduce un nuovo stato iniziale (che è anche finale per accettare ε) con una ε -transizione allo stato iniziale originale, e aggiunge ε -transizioni da tutti gli stati finali originali al nuovo stato iniziale, creando così i cicli necessari.

Il complemento di un linguaggio regolare è regolare. Questa proprietà non è ovvia dalla definizione di espressione regolare, ma è immediata dalla caratterizzazione mediante automi: dato un DFA completo per L (dove "completo" significa che da ogni stato con ogni simbolo esiste una transizione), basta scambiare stati finali con non finali per ottenere un automa per il complemento $\Sigma^* \setminus L$. La completezza è essenziale: se l'automata non è completo, dobbiamo prima aggiungere uno stato "pozzo" che assorbe tutte le transizioni mancanti.

L'intersezione di linguaggi regolari è regolare. Questo può essere dimostrato usando la chiusura rispetto a unione e complemento attraverso la legge di De Morgan: $L_1 \cap L_2 = \text{complemento di } (\text{complemento di } L_1 \cup \text{complemento di } L_2)$. Alternativamente, possiamo costruire direttamente un automa prodotto: dati DFA $A_1 = (Q_1, \Sigma, \delta_1, q_0^1, F_1)$ e $A_2 = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$, l'automata prodotto ha stati $Q_1 \times Q_2$, stato iniziale (q_0^1, q_0^2) , stati finali $F_1 \times F_2$, e transizione $\delta((p_1, p_2), a) = (\delta_1(p_1, a), \delta_2(p_2, a))$. Questo automa accetta esattamente $L_1 \cap L_2$.

La differenza di linguaggi regolari è regolare, poiché $L_1 \setminus L_2 = L_1 \cap (\text{complemento di } L_2)$, combinando intersezione e complemento. Analogamente, la differenza simmetrica $L_1 \oplus L_2 = (L_1 \setminus L_2) \cup (L_2 \setminus L_1)$ è regolare.

Omomorfismi e Sostituzione

I morfismi di monoidi, già introdotti precedentemente, giocano un ruolo centrale nelle proprietà di chiusura. Un morfismo $h: \Sigma^* \rightarrow \Gamma^*$ induce operazioni naturali sui linguaggi. L'immagine diretta preserva la regolarità: se $L \subseteq \Sigma^*$ è regolare, allora $h(L) = \{h(w) : w \in L\} \subseteq \Gamma^*$ è regolare. Questo si dimostra facilmente con le espressioni regolari: data un'espressione r per L , sostituiamo ogni simbolo a con $h(a)$ ottenendo un'espressione per $h(L)$.

Anche l'immagine inversa preserva la regolarità, un fatto meno ovvio ma altrettanto fondamentale. Se $K \subseteq \Gamma^*$ è regolare e $h: \Sigma^* \rightarrow \Gamma^*$ è un morfismo, allora $h^{-1}(K) = \{w \in \Sigma^* : h(w) \in K\}$ è regolare. La dimostrazione mediante automi è costruttiva: dato un DFA $A = (Q, \Gamma, \delta, q_0, F)$ per K , costruiamo un automa per $h^{-1}(K)$ con gli stessi stati Q , ma funzione di transizione modificata: $\delta'(q, a) = \delta(q, h(a))$, ovvero seguiamo in A il cammino etichettato da $h(a)$. Questo automa accetta w se e solo se il cammino etichettato da $h(w)$ in A porta ad uno stato finale, ovvero se e solo se $h(w) \in K$.

Una generalizzazione potente dei morfismi è la sostituzione. Una sostituzione σ associa a ogni simbolo $a \in \Sigma$ un linguaggio $\sigma(a) \subseteq \Gamma^*$, ed estende a stringhe mediante $\sigma(a_1 a_2 \dots a_n) = \sigma(a_1) \sigma(a_2) \dots \sigma(a_n)$, dove la concatenazione è tra linguaggi. Se $L \subseteq \Sigma^*$ è un linguaggio, $\sigma(L) = \bigcup \{w \in L\} \sigma(w)$.

Il teorema sulla sostituzione afferma che se L è regolare e $\sigma(a)$ è regolare per ogni $a \in \Sigma$, allora $\sigma(L)$ è regolare. La dimostrazione procede per induzione strutturale sulle espressioni regolari. I casi base sono immediati: $\sigma(\emptyset) = \emptyset$ è regolare, $\sigma(\varepsilon) = \{\varepsilon\}$ è regolare, $\sigma(a)$ è regolare per ipotesi. Per i casi induttivi: $\sigma(r + s) = \sigma(r) \cup \sigma(s)$ è regolare per chiusura rispetto all'unione, $\sigma(rs) = \sigma(r)\sigma(s)$ è regolare per chiusura rispetto alla concatenazione, e $\sigma(r^*) = (\sigma(r))^*$ è regolare per chiusura rispetto alla stella.

Quoziente e Derivata di Linguaggi

Il quoziente di un linguaggio L rispetto a una stringa u , denotato $u^{-1}L$ o L/u , è definito come $\{w \in \Sigma^* : uw \in L\}$, l'insieme delle stringhe che, quando precedute da u , producono stringhe in L . Se L è regolare, allora $u^{-1}L$ è regolare per ogni u . Questo segue dalla caratterizzazione mediante automi: dato un DFA per L con stato iniziale q_0 , l'automa per $u^{-1}L$ ha come stato iniziale $\delta(q_0, u)$, ovvero lo stato raggiunto dopo aver letto u .

Il quoziente può essere generalizzato a linguaggi: dati $L_1, L_2 \subseteq \Sigma^*$, il quoziente sinistro è $L_1^{-1}L_2 = \{w : \exists u \in L_1. uw \in L_2\}$. Se L_1 e L_2 sono regolari, $L_1^{-1}L_2$ è regolare. Questo si dimostra scrivendo $L_1^{-1}L_2 = \bigcup \{u \in L_1\} u^{-1}L_2$, e osservando che l'unione infinita di linguaggi regolari non è necessariamente regolare, ma in questo caso particolare possiamo usare la costruzione prodotto sugli automi per ottenere direttamente un automa per $L_1^{-1}L_2$.

La derivata di Brzozowski generalizza il quoziente a un'operazione algebrica sulle espressioni regolari. Data un'espressione regolare r e un simbolo a , la derivata $\partial_a r$ è un'espressione regolare tale che $L(\partial_a r) = a^{-1}L(r)$. Le regole di calcolo sono:

$\partial_a \emptyset = \emptyset$ $\partial_a \varepsilon = \emptyset$ $\partial_a b = \varepsilon$ se $a = b$, altrimenti $\emptyset$ $\partial_a(r + s) = \partial_a r + \partial_a s$ $\partial_a(rs) = (\partial_a r)s + v(r)\partial_a s$, dove $v(r) = \varepsilon$ se $\varepsilon \in L(r)$, altrimenti $\emptyset$ $\partial_a(r^*) = (\partial_a r)r^*$

Queste regole permettono di calcolare simbolicamente la derivata di qualsiasi espressione regolare. La derivata rispetto a una stringa $w = a_1 a_2 \dots a_n$ si ottiene componendo le derivate rispetto ai singoli simboli: $\partial_w r = \partial_{a_n} \dots \partial_{a_2} \partial_{a_1} r$.

L'insieme delle derivate di un'espressione regolare r è finito. Questo fatto notevole implica che possiamo costruire un automa per $L(r)$ i cui stati sono le derivate di r : lo stato iniziale è r , gli stati finali sono le derivate s tali che $\varepsilon \in L(s)$, e la transizione da s con simbolo a porta a $\partial_a s$. Questo automa, chiamato automa delle derivate, riconosce esattamente $L(r)$ e ha al più tante stati quante sono le derivate distinte di r .

La finitezza dell'insieme delle derivate si dimostra osservando che ogni derivata di r è un'espressione costruita a partire dalle sottoespressioni di r usando solo unione, concatenazione e stella. Poiché r ha un numero finito di sottoespressioni, l'insieme delle possibili derivate è finito. Brzozowski mostrò che questo automa delle derivate è spesso vicino al minimo, anche se non necessariamente minimale.

Chiusura Rispetto a Operazioni Avanzate

Consideriamo ora operazioni più sofisticate. Il reversal (inversione) di una stringa $w = a_1a_2\dots a_n$ è $w^R = a_n a_{n-1} \dots a_1$. Il reversal di un linguaggio è $L^R = \{w^R : w \in L\}$. Se L è regolare, L^R è regolare. Questo si vede facilmente mediante NFA: dato un NFA per L , invertiamo la direzione di tutte le transizioni, scambiamo il ruolo dello stato iniziale con quello degli stati finali (creando un nuovo stato finale unico se ce ne sono più di uno), ottenendo un NFA per L^R .

L'operazione di shuffle (interleaving) combina due stringhe inserendo i loro simboli in modo alternato o più generalmente in qualsiasi ordine che preservi l'ordine relativo all'interno di ciascuna stringa. Formalmente, $u \sqcup v = \{w : w = u_1v_1u_2v_2\dots u_nv_n \text{ dove } u = u_1u_2\dots u_n \text{ e } v = v_1v_2\dots v_n\}$. Per linguaggi, $L_1 \sqcup L_2 = \{u \in L_1, v \in L_2\} u \sqcup v$. I linguaggi regolari sono chiusi rispetto allo shuffle: se L_1 e L_2 sono regolari, $L_1 \sqcup L_2$ è regolare. La dimostrazione usa automi prodotto con transizioni non deterministiche che scelgono da quale automa leggere il prossimo simbolo.

La proiezione ciclica di una stringa $w = a_1a_2\dots a_n$ è l'insieme delle sue rotazioni cicliche: $\{a_1a_2\dots a_n, a_2a_3\dots a_na_1, \dots, a_na_1a_2\dots a_{n-1}\}$. Per un linguaggio L , la chiusura ciclica è $\text{Cyclic}(L) = \{w \in L\} \cup \{\text{rotazioni di } w\}$. I linguaggi regolari sono chiusi rispetto alla chiusura ciclica. Un modo di vederlo è osservare che w è una rotazione ciclica di w' se e solo se $w \in \Sigma uv$ e $w' \in \Sigma vu$ per qualche u, v tali che $w' = uv$. Quindi $\text{Cyclic}(L)$ può essere espresso usando operazioni regolari e quozienti.

Equivalenza e Decidibilità

Problemi di Decisione Fondamentali

Un aspetto cruciale della teoria dei linguaggi regolari è la decidibilità di molti problemi naturali. Questa decidibilità contrasta fortemente con la situazione per classi di linguaggi più generali, dove molti problemi fondamentali sono indecidibili.

Il problema del vuoto chiede se un dato linguaggio regolare è vuoto, ovvero se $L = \emptyset$. Questo problema è decidibile in tempo lineare nella dimensione della rappresentazione (automa o espressione regolare). Per un DFA, basta verificare se esiste un cammino dallo stato iniziale a qualche stato finale, il che si può fare con una visita in ampiezza o profondità del grafo delle transizioni. Per un'espressione regolare, si può calcolare ricorsivamente se $\varepsilon \in L(r)$, tenendo conto che $L(r) = \emptyset$ se e solo se r si semplifica a $\emptyset$ usando le leggi algebriche.

Il problema dell'appartenenza chiede se una data stringa w appartiene a un dato linguaggio regolare L . Per un DFA, questo si risolve simulando l'automa sulla stringa w e verificando se termina in uno stato finale. La complessità temporale è $O(|w|)$, lineare nella lunghezza della stringa. Per un NFA, dobbiamo tracciare l'insieme di tutti gli stati raggiungibili, che nel caso peggiore può richiedere tempo $O(|w| \cdot |Q|^2)$ con approcci ingenui, ma può essere ottimizzato a $O(|w| \cdot |Q|)$ mantenendo esplicitamente l'insieme degli stati correnti.

Il problema dell'equivalenza chiede se due linguaggi regolari L_1 e L_2 sono uguali. Questo problema è decidibile usando la chiusura dei linguaggi regolari rispetto a operazioni

booleane. Due linguaggi sono equivalenti se e solo se la loro differenza simmetrica è vuota: $L_1 = L_2 \Leftrightarrow (L_1 \setminus L_2) \cup (L_2 \setminus L_1) = \emptyset$. Dato che possiamo costruire automi per $L_1 \setminus L_2$ e $L_2 \setminus L_1$, e verificare il vuoto, l'equivalenza è decidibile.

Un algoritmo più efficiente per l'equivalenza si basa sulla minimizzazione. Costruiamo i DFA minimi per L_1 e L_2 . Due linguaggi sono equivalenti se e solo se i loro DFA minimi sono isomorfi (a meno di ridenominazione degli stati). Poiché il DFA minimo è unico, basta verificare che abbiano lo stesso numero di stati e la stessa struttura di transizioni. La complessità totale è dominata dalla minimizzazione, che richiede tempo $O(n \log n)$ con l'algoritmo di Hopcroft.

Il problema dell'inclusione chiede se $L_1 \subseteq L_2$. Questo è equivalente a verificare se $L_1 \setminus L_2 = \emptyset$, quindi è decidibile con lo stesso metodo del vuoto. La complessità è polinomiale nella dimensione degli automi.

Il problema dell'universalità chiede se $L = \Sigma^*$, ovvero se il linguaggio contiene tutte le stringhe possibili. Questo è equivalente a verificare se il complemento è vuoto: $L = \Sigma^* \Leftrightarrow \Sigma^* \setminus L = \emptyset$. Quindi anche questo problema è decidibile in tempo polinomiale.

Il problema della finitudine chiede se un linguaggio regolare è finito o infinito. Un linguaggio regolare è infinito se e solo se esiste una stringa $w \in L$ la cui lunghezza è compresa tra n e $2n - 1$, dove n è il numero di stati del DFA minimo. Questo segue dal pumping lemma: se esistono stringhe arbitrariamente lunghe in L , allora per il pumping lemma esiste una stringa pumpabile, e quindi stringhe di tutte le lunghezze sufficientemente grandi. L'algoritmo verifica semplicemente se esistono stringhe in questo intervallo di lunghezze, il che può essere fatto con una ricerca esaustiva o analizzando i cicli nel DFA.

Complessità Computazionale dei Problemi di Decisione

Sebbene tutti i problemi menzionati siano decidibili, è interessante analizzare la loro complessità computazionale precisa. Il problema del vuoto è in NLOGSPACE (spazio logaritmico non deterministico), poiché possiamo verificare l'esistenza di un cammino dallo stato iniziale a uno stato finale usando solo spazio logaritmico per memorizzare lo stato corrente.

Il problema dell'equivalenza è più complesso. Per DFA, è in PSPACE (spazio polinomiale), ma si congettura che sia più facile. Esiste un algoritmo che richiede tempo $O(n^2)$ dove n è il numero totale di stati dei due automi, basato su una variante dell'algoritmo di Hopcroft. Per NFA, il problema dell'equivalenza è PSPACE-completo, il che significa che è intrinsecamente più difficile del caso deterministico a meno che $PSPACE = P$, il che è ritenuto improbabile.

La differenza di complessità tra DFA e NFA riflette il fatto che il non determinismo può compattare esponenzialmente l'informazione. Verificare l'equivalenza di NFA richiede essenzialmente di esplorarli completamente, il che può richiedere spazio polinomiale nel caso peggiore.

Il problema della minimizzazione di un DFA è in P (tempo polinomiale), con complessità $O(n \log n)$ usando l'algoritmo di Hopcroft. Per NFA, come già menzionato, la minimizzazione è PSPACE-completa, quindi presumibilmente intrattabile.

Algoritmi di Equivalenza Avanzati

Oltre all'approccio basato sulla minimizzazione, esistono algoritmi alternativi per verificare l'equivalenza di automi che possono essere più efficienti in certi casi pratici. L'algoritmo di Hopcroft-Karp usa una tecnica di raffinamento di partizioni simile alla minimizzazione, ma opera simultaneamente sui due automi da confrontare.

L'idea è di mantenere una partizione dello spazio prodotto $Q_1 \times Q_2$, dove Q_1 e Q_2 sono gli insiemi di stati dei due automi. Inizialmente, la partizione separa le coppie (q_1, q_2) dove q_1 è finale in A_1 e q_2 non è finale in A_2 (o viceversa) dalle altre coppie. Poi si raffina iterativamente la partizione separando coppie che con lo stesso simbolo transitano in classi diverse.

Se alla fine la coppia (q_0^1, q_0^2) degli stati iniziali appartiene a una classe insieme a una coppia dove uno stato è finale e l'altro no, allora i linguaggi sono diversi. Altrimenti sono equivalenti. Questo algoritmo può terminare prima della minimizzazione completa se scopre una differenza, ed è quindi talvolta più efficiente in pratica.

Un approccio completamente diverso usa la programmazione dinamica per calcolare la funzione di equivalenza. Definiamo $E_k(p, q)$ come vero se e solo se gli stati p di A_1 e q di A_2 accettano le stesse stringhe di lunghezza al più k . Chiaramente $E_0(p, q)$ è vero se e solo se p e q sono entrambi finali o entrambi non finali. Ricorsivamente, $E_{k+1}(p, q)$ è vero se e solo se $E_k(p, q)$ è vero e per ogni simbolo a , $E_k(\delta_1(p, a), \delta_2(q, a))$ è vero.

Calcoliamo E_k per $k = 0, 1, 2, \dots$ fino a quando si stabilizza (il che deve accadere entro $n_1 + n_2$ passi, dove n_1 e n_2 sono i numeri di stati). I linguaggi sono equivalenti se e solo se $E_{\{n_1+n_2\}}(q_0^1, q_0^2)$ è vero. Questo algoritmo ha complessità $O((n_1 + n_2)^2 |Σ|)$ in tempo e $O(n_1 \cdot n_2)$ in spazio.

La Gerarchia di Chomsky e Oltre

Linguaggi Context-Free e Automi a Pila

I linguaggi regolari costituiscono il livello più basso della gerarchia di Chomsky, che classifica i linguaggi formali in base alla complessità delle grammatiche che li generano. Il livello successivo è quello dei linguaggi context-free, che possono essere generati da grammatiche context-free (CFG) o riconosciuti da automi a pila (PDA).

Un automa a pila è simile a un automa a stati finiti ma dotato di una memoria ausiliaria strutturata come pila (stack). Formalmente, un PDA è una settupla $P = (Q, Σ, Γ, δ, q_0, Z_0, F)$ dove Q è l'insieme degli stati, $Σ$ è l'alfabeto di input, $Γ$ è l'alfabeto della pila, $δ: Q \times (Σ \cup \{\epsilon\}) \times Γ \rightarrow \mathcal{P}(Q \times Γ^*)$ è la funzione di transizione, q_0 è lo stato iniziale, $Z_0 \in Γ$ è il simbolo iniziale sulla pila, e $F \subseteq Q$ è l'insieme degli stati finali.

La transizione $\delta(q, a, Z) = \{(p_1, \alpha_1), (p_2, \alpha_2), \dots\}$ significa che quando l'automa è nello stato q , legge il simbolo a (o ϵ per una transizione spontanea), e vede Z in cima alla pila, può transitare in uno degli stati p_i e sostituire Z con la stringa α_i sulla pila. Se $\alpha_i = \epsilon$, il simbolo viene eliminato (pop), se $\alpha_i = Z$, la pila non cambia, se $\alpha_i = ZW$, viene aggiunto W sopra Z .

I linguaggi context-free sono strettamente più potenti dei linguaggi regolari. Il classico esempio è $L = \{a^n b^n : n \geq 0\}$, che non è regolare (come dimostrato dal pumping lemma) ma è context-free. Un PDA per L usa la pila per contare le a : legge a e le impila, poi quando incontra b inizia a eliminarle dalla pila, una per ogni b letta. Accetta se la pila è vuota alla fine.

Dal punto di vista algebrico, i linguaggi context-free non hanno una caratterizzazione così elegante come quella dei linguaggi regolari in termini di semigrupp sintattici. Tuttavia, esistono strutture algebriche più complesse, come i magmoidi, che catturano parzialmente la struttura dei linguaggi context-free. Un magmoide è un insieme dotato di un'operazione binaria parziale, generalizzando l'idea di semigruppo.

Linguaggi Context-Sensitive e Macchine di Turing Limitate

Il terzo livello della gerarchia di Chomsky consiste dei linguaggi context-sensitive, generati da grammatiche context-sensitive o riconosciuti da automi linearly bounded (LBA), che sono macchine di Turing la cui nastro è limitato a essere lungo al più un multiplo costante dell'input.

Un esempio di linguaggio context-sensitive che non è context-free è $L = \{a^n b^n c^n : n \geq 0\}$. Questo linguaggio richiede di "contare" simultaneamente tre tipi di simboli, il che supera le capacità di un automa a pila con una sola pila. Un LBA può però gestirlo usando il nastro per memorizzare contatori per ciascun tipo di simbolo.

I linguaggi context-sensitive sono chiusi rispetto a unione, intersezione e complemento, ma la dimostrazione della chiusura rispetto al complemento è molto più difficile che per i linguaggi regolari, ed è stata ottenuta solo negli anni Ottanta da Immerman e Szelepcsényi indipendentemente.

Linguaggi Ricorsivamente Enumerabili e Limiti della Computabilità

Il quarto livello della gerarchia di Chomsky comprende tutti i linguaggi ricorsivamente enumerabili, ovvero quelli riconoscibili da una macchina di Turing senza restrizioni. Questo è il livello più generale di linguaggi che possono essere riconosciuti algoritmicamente, ma include linguaggi per cui l'algoritmo di riconoscimento potrebbe non terminare su alcuni input.

Esistono linguaggi che non sono ricorsivamente enumerabili. Il più famoso è il complemento del problema dell'arresto: l'insieme delle coppie (M, w) dove M è una macchina di Turing che non termina su input w . Questo insieme non è ricorsivamente enumerabile, il che si dimostra con un argomento diagonale simile a quello usato da Cantor per dimostrare la non numerabilità dei numeri reali.

La gerarchia di Chomsky fornisce una scala di complessità crescente, ma all'interno di ciascun livello esistono ulteriori distinzioni. Per i linguaggi regolari, abbiamo visto classificazioni basate su proprietà algebriche dei semigrupp sintattici. Per i linguaggi context-free, esistono sottoclassi importanti come i linguaggi deterministici context-free, riconoscibili da automi a pila deterministici, e i linguaggi $LL(k)$ e $LR(k)$, importanti per la compilazione.

Teoria dei Codici e Proprietà Combinatorie

Codici Liberi e Decomposizione Unica

La teoria dei codici studia rappresentazioni univoche di informazione mediante stringhe. Un sottoinsieme X di Σ^* si dice codice se ogni stringa in X^* ha una decomposizione unica come concatenazione di elementi di X . Formalmente, X è un codice se per ogni $n, m \geq 1$ e $x_1, \dots, x_n, y_1, \dots, y_m \in X$, se $x_1 \dots x_n = y_1 \dots y_m$ allora $n = m$ e $x_i = y_i$ per ogni i .

La condizione di codice è equivalente a richiedere che il morfismo canonico $\varphi: \{1, 2, \dots, k\}^* \rightarrow \Sigma^*$ definito da $\varphi(i) = x_i$ (dove $X = \{x_1, \dots, x_k\}$) sia iniettivo. Un codice permette quindi di rappresentare sequenze di "messaggi" (elementi di X) come stringhe in modo non ambiguo.

Un codice X si dice prefisso se nessun elemento di X è prefisso proprio di un altro elemento di X . I codici prefisso sono importanti perché permettono decodifica istantanea: possiamo decodificare una stringa leggendola da sinistra a destra e identificando ciascun elemento di X non appena lo incontriamo, senza dover guardare avanti. Esempi di codici prefisso includono i codici di Huffman usati per la compressione dati.

Il teorema di Sardinas-Patterson fornisce un algoritmo per verificare se un insieme finito X è un codice. L'algoritmo costruisce iterativamente una sequenza di insiemi $U_0, U_1, U_2, \dots$ dove U_0 contiene tutti i suffissi propri di elementi di X^2 (concatenazioni di due elementi di X), e U_{i+1} contiene i suffissi propri degli elementi in XU_i e U_iX . L'insieme X è un codice se e solo se ε non appare mai in alcun U_i .

La dimostrazione della correttezza si basa sull'osservazione che se X non è un codice, esiste una "sovrapposizione critica" dove due decomposizioni diverse dello stesso prefisso di una stringa divergono. Gli insiemi U_i catturano sistematicamente tutte le possibili sovrapposizioni di elementi di X a profondità crescente.

Un risultato fondamentale dovuto a Schützenberger afferma che un sottoinsieme finito X di Σ^* è un codice se e solo se esiste un automa non ambiguo che riconosce X^* . Un automa è non ambiguo se ogni stringa accettata ha esattamente un cammino accettante. Questo collega la teoria dei codici alla teoria degli automi in modo profondo.

Il Teorema di McMillan e Disuguaglianza di Kraft

Il teorema di McMillan, fondamentale nella teoria dell'informazione, caratterizza i codici istantanei (sinonimi di codici prefisso) in termini di una disuguaglianza sulle lunghezze delle parole di codice. Se $X = \{x_1, \dots, x_n\}$ è un codice prefisso su un alfabeto di cardinalità r , allora la disuguaglianza di Kraft è soddisfatta:

$$\sum_i r^{-|x_i|} \leq 1$$

Reciprocamente, se abbiamo lunghezze $l_1, \dots, l_n$ che soddisfano la disuguaglianza di Kraft, esiste un codice prefisso con queste lunghezze. Questo teorema è cruciale nella progettazione di codici ottimali: ci permette di determinare se un dato schema di lunghezze è realizzabile come codice prefisso senza dover costruire esplicitamente il codice.

La dimostrazione della disuguaglianza di Kraft usa un argomento combinatorio elegante. Visualizziamo il codice come un albero r -ario dove ogni parola di codice corrisponde a una foglia. Un prefisso di lunghezza l "blocca" r^{d-l} foglie a profondità d , dove d è la profondità massima dell'albero. La condizione che nessuna parola sia prefisso di un'altra significa che i sottoinsiemi di foglie bloccati sono disgiunti. Sommando su tutte le parole di codice e normalizzando per il numero totale di foglie r^d , otteniamo la disuguaglianza.

Il teorema di McMillan estende questo risultato ai codici che non sono necessariamente prefisso: anche per codici generici (non prefisso), la disuguaglianza di Kraft deve essere soddisfatta. Questo risultato sorprendente mostra che i codici prefisso sono ottimali rispetto a tutti gli altri codici nel senso che non esistono codici con lunghezze di parole più corte che non potrebbero essere realizzate anche come codici prefisso.

Codici Massimali e Completezza

Un codice X si dice massimale se non è contenuto propriamente in alcun altro codice, ovvero se aggiungere qualsiasi stringa a X distrugge la proprietà di codice. I codici massimali hanno la proprietà che ogni stringa sufficientemente lunga può essere decodificata fino a un certo punto usando elementi di X .

Un codice prefisso X si dice completo se ogni stringa infinita su Σ ha un prefisso in X^* . Equivalentemente, X è completo se la disuguaglianza di Kraft è un'uguaglianza. I codici completi sono massimali: se un codice prefisso è completo, non possiamo aggiungere alcuna stringa senza violare la proprietà di codice o la disuguaglianza di Kraft.

La completezza è importante nella teoria dell'informazione perché i codici completi massimizzano l'efficienza: non "sprecano" alcun spazio nel senso della disuguaglianza di Kraft. I codici di Huffman, per esempio, sono sempre completi quando costruiti correttamente.

Proprietà di Scambio e Teoremi Combinatori

La teoria combinatoria delle stringhe studia proprietà profonde che emergono dalle relazioni tra stringhe e dalle loro strutture di sovrapposizione. Uno dei risultati più eleganti è il teorema di Fine e Wilf, già menzionato precedentemente, che caratterizza quando una stringa con due periodi deve avere un periodo comune più piccolo.

Il lemma di Lyndon-Schützenberger, anch'esso già citato, afferma che se $x^m = y^n$ con $m, n \geq 2$, allora x e y sono potenze di una stringa comune. La dimostrazione è un bellissimo esempio di ragionamento combinatorio. Supponiamo $|x| \leq |y|$ senza perdita di generalità. Allora y inizia con x , ovvero $y = xz$ per qualche stringa z . Sostituendo nell'equazione originale: $x^m = (xz)^n = x(zx)^{n-1}zx\dots$

Procedendo con analisi combinatoria attenta sui prefissi e usando il fatto che $m, n \geq 2$, si può mostrare che x e z devono commutare ($xz = zx$), il che implica che sono entrambi potenze di una stringa primitiva comune per il teorema fondamentale sulla commutazione delle stringhe.

Questi risultati hanno applicazioni nella teoria degli automi, nella teoria dei linguaggi formali, e persino in rami apparentemente non correlati come la topologia algebrica e la teoria dei nodi, dove le relazioni di commutazione tra generatori di gruppi possono essere studiate usando tecniche combinatorie su stringhe.

Automi Bidimensionali e Generalizzazioni

Automi su Alberi e Strutture Gerarchiche

Mentre gli automi a stati finiti standard operano su stringhe (strutture lineari), molte generalizzazioni considerano strutture più complesse. Gli automi su alberi processano alberi ordinati etichettati, strutture fondamentali nella rappresentazione di dati gerarchici come documenti XML, alberi sintattici di programmi, e termini algebrici.

Un automa ad albero bottom-up è una tupla $A = (Q, \Sigma, \delta, F)$ dove Q è un insieme finito di stati, Σ è un alfabeto classificato (ranked alphabet) dove ogni simbolo ha un'arità, δ è un insieme di regole di transizione della forma $f(q_1, \dots, q_k) \rightarrow q$ dove f è un simbolo di arità k , e $F \subseteq Q$ è l'insieme degli stati finali.

L'automa processa un albero a partire dalle foglie, assegnando stati ai nodi usando le regole di transizione. Un albero è accettato se la radice riceve uno stato finale. Questa nozione generalizza naturalmente gli automi a stati finiti: una stringa può essere vista come un albero degenerato dove ogni nodo interno ha esattamente un figlio.

Il teorema di Thatcher e Wright afferma che i linguaggi di alberi riconoscibili da automi ad albero sono esattamente quelli chiusi rispetto alla congruenza di Myhill-Nerode generalizzata agli alberi. Inoltre, questi linguaggi sono esattamente quelli definibili in logica monadica del second'ordine sugli alberi, estendendo il teorema di Büchi-Elgot-Trakhtenbrot.

Gli automi ad albero hanno applicazioni profonde in logica, teoria dei tipi, e verifica formale. Il model checking di formule logiche su strutture ad albero può essere ridotto al problema di vuoto per automi ad albero, permettendo algoritmi efficienti per verificare proprietà di programmi e sistemi.

Automi Alternanti e Giochi

Gli automi alternanti introducono una forma duale del non determinismo: oltre a transizioni esistenziali (l'automa può scegliere tra più possibilità), permettono transizioni universali (l'automa deve esplorare tutte le possibilità simultaneamente). Formalmente, la funzione di transizione di un automa alternante mappa (stato, simbolo) a una formula booleana su stati, usando AND e OR.

Un esempio di transizione alternante è $\delta(q, a) = (q_1 \wedge q_2) \vee q_3$, che significa: leggendo a dallo stato q , l'automa deve transitare sia in q_1 che in q_2 , oppure può transitare in q_3 . L'accettazione richiede che esista una "strategia" che porta sempre a stati finali, indipendentemente dalle scelte non deterministiche.

Gli automi alternanti hanno la stessa potenza espressiva degli automi non deterministici per linguaggi di stringhe e alberi regolari, ma possono essere esponenzialmente più concisi. La conversione da automi alternanti ad automi non deterministici usa una costruzione prodotto che crea uno stato per ogni sottoinsieme di stati dell'automa alternante, simile alla costruzione per sottoinsiemi ma più complessa a causa della struttura AND-OR.

La semantica degli automi alternanti può essere interpretata in termini di giochi tra due giocatori: un giocatore cerca di accettare (Proponent) e l'altro cerca di rifiutare (Opponent). Le transizioni OR sono mosse del Proponent, le AND sono mosse dell'Opponent. Il Proponent vince se esiste una strategia che garantisce di raggiungere stati finali indipendentemente dalle mosse dell'Opponent.

Questa interpretazione in termini di giochi collega gli automi alla teoria dei giochi e alla logica temporale, dove formule modali possono essere valutate come giochi tra un verificatore e un falsificatore.

Automi su Parole Infinite e ω -Automi

Gli automi su parole infinite, o ω -automi, processano sequenze infinite di simboli. Questi sono cruciali per modellare e verificare sistemi reattivi, processi non terminanti, e comportamenti continui. La definizione è simile agli automi finiti standard, ma la condizione di accettazione è modificata per tener conto della natura infinita dell'input.

Un automa di Büchi ha una condizione di accettazione che richiede che almeno uno stato finale sia visitato infinitamente spesso durante una computazione infinita. Formalmente, una computazione infinita $q_0, q_1, q_2, \dots$ è accettante se l'insieme $\{i : q_i \in F\}$ è infinito.

Altri tipi di ω -automi includono automi di Muller (accettanti se l'insieme degli stati visitati infinitamente spesso appartiene a una famiglia specificata), automi di Rabin e Streett (con condizioni di accettazione basate su coppie di insiemi di stati), e automi di parità (dove ogni stato ha una priorità e l'accettazione dipende dalla parità della priorità minima vista infinitamente spesso).

Questi diversi tipi di ω -automi hanno relazioni di potenza espressiva complesse. Gli automi di Büchi non deterministici e gli automi di Muller hanno la stessa potenza espressiva e riconoscono esattamente gli ω -linguaggi regolari. Tuttavia, gli automi di Büchi deterministici sono strettamente meno potenti: non tutti gli ω -linguaggi regolari possono essere riconosciuti da automi di Büchi deterministici.

Gli automi di Rabin, Streett e parità deterministici hanno tutti la stessa potenza degli automi di Büchi non deterministici. Le conversioni tra questi formati sono algoritmiche ma possono causare incrementi esponenziali nella dimensione dell'automa.

Il teorema di Büchi per ω -linguaggi afferma che un ω -linguaggio è riconoscibile da un automa di Büchi se e solo se è definibile in logica monadica del second'ordine sull'ordine $\omega = \{0, 1, 2, \dots\}$. Questo estende il teorema per stringhe finite e fornisce una caratterizzazione logica completa degli ω -linguaggi regolari.

Applicazioni Pratiche e Computazionali

Espressioni Regolari in Informatica Pratica

Le espressioni regolari sono uno degli strumenti più utilizzati in informatica pratica, con applicazioni che spaziano dalla validazione di input, al text processing, all'analisi lessicale nei compilatori. Tuttavia, le espressioni regolari usate in linguaggi di programmazione moderni come Perl, Python, Java e JavaScript spesso includono estensioni che vanno oltre il formalismo teorico delle espressioni regolari classiche.

Queste estensioni includono backreferences (riferimenti all'indietro a sottostringhe precedentemente matchate), lookahead e lookbehind assertions (asserzioni su ciò che segue o precede senza consumare caratteri), e recursive patterns. Alcune di queste estensioni, in particolare i backreferences, aumentano la potenza espressiva oltre i linguaggi regolari: l'espressione $(a^+)b^1$ che matcha a^nb^n riconosce un linguaggio non regolare.

La conseguenza pratica è che l'implementazione di questi motori di espressioni regolari estese non può usare automi a stati finiti standard. Invece, usano algoritmi di backtracking che possono avere complessità esponenziale nel caso peggiore. Questo ha portato a problemi di sicurezza noti come "ReDoS" (Regular Expression Denial of Service), dove espressioni regolari mal scritte possono causare tempi di esecuzione esponenziali su certi input malevoli.

Gli algoritmi moderni per matching di espressioni regolari classiche (senza estensioni) usano tecniche sofisticate per evitare backtracking. L'algoritmo di Thompson costruisce un NFA dall'espressione regolare in tempo lineare, poi usa una simulazione del NFA che mantiene esplicitamente l'insieme degli stati correnti. Questo garantisce complessità $O(mn)$ dove m è la dimensione dell'espressione e n la lunghezza dell'input.

Un'alternativa è l'algoritmo di Glushkov, che costruisce un automa dalle "posizioni" nell'espressione regolare. Ogni occorrenza di un simbolo nell'espressione riceve un'etichetta unica, e l'automa traccia quali posizioni sono state matchate. Questo produce automi con al più $m + 1$ stati dove m è la dimensione dell'espressione, spesso più piccoli dell'automa di Thompson.

Compilatori e Analisi Lessicale

L'analisi lessicale (lexing) è la prima fase della compilazione, dove il codice sorgente viene segmentato in token (parole chiave, identificatori, operatori, letterali). Gli analizzatori lessicali sono implementati quasi universalmente usando automi a stati finiti derivati da espressioni regolari.

Il generatore di scanner LEX (e la sua versione GNU, flex) accetta in input una specifica di token come insieme di espressioni regolari, e genera codice C per un automa deterministico che riconosce questi token. Il processo include:

1. Conversione di ciascuna espressione regolare in un NFA usando la costruzione di Thompson
2. Combinazione degli NFA in un unico NFA che riconosce l'unione di tutti i token
3. Conversione dell'NFA combinato in un DFA usando la costruzione per sottoinsiemi
4. Minimizzazione del DFA usando l'algoritmo di Hopcroft
5. Generazione di codice C che implementa il DFA minimizzato

La sfida principale nell'analisi lessicale è gestire l'ambiguità quando più token potrebbero matchare un prefisso dell'input. La convenzione standard è la regola del "longest match" (matcha la sottostringa più lunga possibile) combinata con la regola della priorità (se più pattern matchano la stessa lunghezza, usa il primo definito).

Queste regole possono essere implementate efficientemente nel DFA: si continua a leggere caratteri finché il DFA accetta, memorizzando l'ultimo stato accettante e la posizione corrispondente, poi si ritorna a quella posizione quando il DFA non può più procedere.

Verifica Formale e Model Checking

Il model checking è una tecnica automatica per verificare che un sistema soddisfi certe proprietà specificate formalmente. Gli automi a stati finiti giocano un ruolo centrale: il sistema viene modellato come un automa che rappresenta tutti i possibili comportamenti, e le proprietà sono espresse in logica temporale o direttamente come automi.

L'algoritmo fondamentale verifica l'inclusione del linguaggio: dato un automa M che rappresenta il sistema e un automa S che rappresenta la specifica (comportamenti desiderati), verifichiamo se $L(M) \subseteq L(S)$. Questo si riduce a verificare se $L(M) \cap \text{complemento}(L(S)) = \emptyset$, ovvero se non esistono comportamenti del sistema che violano la specifica.

Per sistemi reattivi non terminanti, usiamo ω -automi. La proprietà da verificare è tipicamente espressa in logica temporale lineare (LTL) o computation tree logic (CTL). L'algoritmo di Vardi-Wolper converte formule LTL in automi di Büchi, permettendo di ridurre il model checking LTL alla verifica di vuoto per automi di Büchi.

Un aspetto cruciale è che il numero di stati dell'automa prodotto può essere esponenziale nella dimensione della formula LTL. Questo fenomeno, noto come "state explosion", è la principale limitazione pratica del model checking. Tecniche come abstract interpretation, partial order reduction, e symbolic model checking usando Binary Decision Diagrams (BDD) sono state sviluppate per mitigare questo problema.

Il successo del model checking nell'industria, particolarmente nella verifica di hardware e protocolli di comunicazione, dimostra che nonostante i limiti teorici, gli automi a stati finiti forniscono un formalismo computazionalmente trattabile per molti problemi pratici di verifica.

Algoritmi su Stringhe e Pattern Matching

La teoria degli automi fornisce fondamenti teorici per algoritmi efficienti di pattern matching. L'algoritmo di Knuth-Morris-Pratt (KMP) per cercare un pattern in un testo può essere visto come la simulazione di un automa deterministico costruito dal pattern.

Dato un pattern P di lunghezza m , l'algoritmo KMP precalcola una "funzione di fallimento" che indica, per ogni prefisso di P , quale è il più lungo prefisso proprio che è anche suffisso. Questa funzione corrisponde esattamente alle transizioni di un automa che riconosce Σ^*P (tutte le stringhe che terminano con P).

L'algoritmo di Boyer-Moore, ancora più efficiente in pratica, usa euristiche di "bad character" e "good suffix" che permettono di saltare porzioni del testo senza esaminarle. Anche queste euristiche hanno interpretazioni in termini di automi, sebbene l'automa sottostante non sia costruito esplicitamente.

Per pattern matching con espressioni regolari, l'algoritmo di Aho-Corasick generalizza KMP a insiemi di pattern. Costruisce un automa che riconosce $\Sigma^*(P_1 \cup P_2 \cup \dots \cup P_k)$ dove $P_1, \dots, P_k$ sono i pattern. L'automa ha una struttura trie per i pattern con aggiunta di "failure links" che permettono transizioni efficienti quando un match parziale fallisce.

Questi algoritmi dimostrano come concetti teorici della teoria degli automi si traducano direttamente in algoritmi pratici con garanzie di complessità ottimale. L'algoritmo KMP, per esempio, garantisce tempo $O(n + m)$ per cercare un pattern di lunghezza m in un testo di lunghezza n , che è ottimale poiché dobbiamo almeno leggere l'intero testo.

Frontiere della Ricerca e Problemi Aperti

Complessità Descrittiva e Caratterizzazioni Fini

Nonostante decenni di ricerca, rimangono molte questioni aperte sulla caratterizzazione precisa di classi di linguaggi attraverso logiche, algebre e automi. Un problema fondamentale è trovare caratterizzazioni logiche complete per classi al di là dei linguaggi regolari.

Per i linguaggi context-free, non esiste una logica naturale che li caratterizzi completamente analogamente a come MSO caratterizza i linguaggi regolari. Esistono risultati parziali: i linguaggi context-free deterministici hanno caratterizzazioni in termini di logiche temporali con "until" modulato, ma per la classe completa dei linguaggi context-free manca ancora una caratterizzazione logica soddisfacente.

La congettura di Straubing sulla decidibilità dell'appartenenza ai livelli della gerarchia di Straubing-Thérien rimane parzialmente aperta. Mentre i primi livelli sono stati caratterizzati e dimostrati decidibili, per livelli superiori la situazione è meno chiara. Determinare algebricamente se un dato linguaggio regolare appartiene a un livello specifico della gerarchia è un problema di frontiera.

Complessità della Conversione tra Formalismi

La complessità della conversione tra diverse rappresentazioni di linguaggi regolari è un'area di ricerca attiva. Sappiamo che la conversione da NFA a DFA può richiedere crescita esponenziale, e questo limite è stretto. Ma per conversioni tra espressioni regolari, NFA, DFA, e automi minimali, molte questioni rimangono aperte.

Per esempio, data un'espressione regolare di dimensione n , qual è la dimensione del DFA minimo corrispondente nel caso peggiore? Sappiamo che può essere esponenziale, ma caratterizzare esattamente quando questo accade e se esistono classi di espressioni regolari per cui la conversione è più efficiente rimane oggetto di studio.

La conversione da DFA a espressione regolare può produrre espressioni di dimensione esponenziale usando l'eliminazione di stati. Esistono algoritmi alternativi basati sulla decomposizione algebrica che possono produrre espressioni più piccole in certi casi, ma una caratterizzazione completa della complessità descrittiva (quante più concisa può essere un'espressione regolare rispetto a un automa e viceversa) è ancora incompleta.

Automi Infiniti e Giochi Infiniti

La teoria degli automi su parole infinite ha visto sviluppi significativi negli ultimi decenni, ma rimangono questioni fondamentali. Un problema aperto è la complessità della complementazione di automi di Büchi non deterministici. Sappiamo che è possibile complementare tali automi, ma la costruzione ottimale e i limiti inferiori precisi rimangono oggetto di ricerca.

La determinizzazione di automi di Büchi è impossibile in generale (esistono ω -linguaggi riconoscibili da automi di Büchi non deterministici ma non da automi di Büchi deterministici), ma è possibile convertirli in automi di Rabin, Streett o parità deterministici. La complessità ottimale di queste conversioni, sia nel caso peggiore che in classi ristrette di automi, è un'area attiva di ricerca.

Gli automi su strutture infinite più complesse, come alberi infiniti, grafi, e ordini parziali, presentano sfide ulteriori. Il model checking di logiche temporali ad albero (come la μ -calculus) si riduce alla risoluzione di giochi di parità infiniti. Sebbene esistano algoritmi polinomiali per risolvere questi giochi, trovare algoritmi con complessità ottimale (possibilmente lineare o quasi-lineare) rimane un problema aperto importante con conseguenze pratiche per la verifica formale.

Linguaggi Formali e Machine Learning

Un'area emergente è l'intersezione tra teoria dei linguaggi formali e machine learning. L'apprendimento di linguaggi regolari da esempi (grammatical inference) ha una lunga storia, con risultati sia positivi che negativi. L'algoritmo L^* di Angluin apprende automi deterministici usando query di appartenenza e query di equivalenza, ed è stato applicato con successo in verifica e testing.

Recenti sviluppi in deep learning hanno portato a reti neurali che possono approssimare automi a stati finiti e persino automi a pila. I Transformers, architettura dominante in natural language processing, possono essere visti come automi infiniti con attenzione, e

comprendere le loro capacità espressive in termini di teoria dei linguaggi formali è un'area di ricerca attiva.

La domanda "quali linguaggi possono essere appresi da reti neurali ricorrenti?" non ha ancora una risposta completa. Esistono risultati che mostrano che RNN possono approssimare automi regolari, ma la complessità campionaria e le garanzie di generalizzazione sono meno comprese rispetto all'apprendimento statistico classico.

Quantum Automata e Computazione

Gli automi quantistici sono generalizzazioni degli automi finiti che sfruttano fenomeni quantistici come sovrapposizione e entanglement. Un automa quantistico ha stati quantistici (vettori in uno spazio di Hilbert) e transizioni unitarie. L'accettazione è probabilistica, determinata dalla misurazione dello stato finale.

È stato dimostrato che gli automi quantistici possono riconoscere alcuni linguaggi con meno stati rispetto agli automi classici, sfruttando l'interferenza quantistica. Per esempio, esistono linguaggi riconoscibili da automi quantistici con spazio logaritmico che richiederebbero spazio polinomiale classicamente.

Tuttavia, per linguaggi regolari, gli automi quantistici non aumentano la potenza espressiva: i linguaggi riconoscibili da automi quantistici con probabilità di errore limitato sono esattamente i linguaggi regolari. La questione interessante è quindi la complessità: quando e quanto gli automi quantistici possono essere più efficienti di quelli classici?

Questa domanda ha implicazioni profonde per la complessità computazionale quantistica più in generale. Se potessimo caratterizzare completamente i vantaggi degli automi quantistici, potremmo far luce sulla separazione tra classi di complessità classiche e quantistiche come P vs BQP .

L'algebra delle stringhe e la teoria dei linguaggi formali rappresentano una delle intersezioni più ricche tra matematica, logica e informatica. Partendo dalle definizioni elementari di monoidi liberi e concatenazione, abbiamo visto emergere una struttura teorica di straordinaria profondità e bellezza, con connessioni a discipline apparentemente distanti come la teoria dei semigrupp, la logica matematica, la teoria dei giochi, e persino la fisica quantistica. Le caratterizzazioni multiple dei linguaggi regolari – mediante espressioni regolari, automi a stati finiti, logica monadica del second'ordine, e semigrupp sintattici – dimostrano una notevole robustezza e coerenza della teoria. Questa convergenza di prospettive diverse sullo stesso oggetto matematico suggerisce che i linguaggi regolari catturano un concetto fondamentale della computazione e della struttura formale. Le applicazioni pratiche, dalla compilazione alla verifica formale, dal pattern matching al natural language processing, dimostrano che questa teoria non è meramente accademica ma fornisce strumenti concreti per risolvere problemi reali. La tensione tra decidibilità teorica e complessità computazionale pratica continua a stimolare ricerca su algoritmi sempre più efficienti e rappresentazioni sempre più compatte. Guardando al futuro, l'integrazione con approcci probabilistici e quantistici, l'applicazione al machine learning e all'intelligenza artificiale, e l'estensione a strutture dati più complesse (grafi, database, stream di dati) promettono di mantenere la teoria dei linguaggi formali al centro dell'informatica teorica e applicata per i decenni a venire.